编程狂人

Programming Madman

NO.48

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:http://www.tuicool.com/mags/545827f1d91b1411950453cc

欢迎下载推酷客户端体验更多阅读乐趣

目录

HTML 5 标准定了，哪家欢乐哪家愁

作者：马婧

10月28日，万维网联盟(W3C)宣布HTML5标准最终制定完成并对外发布，这对于HTML5来说有着里程碑似的意义。这也意味着，Native App和Web App之争即将尘埃落定。

一个标准的制定为什么耗时8年之久？表面看来，W3C在推动HTML5前进，但背后究竟是谁裁定了HTML5的命运？为什么说，标准的完工是几家欢喜几家愁？HTML5标准完工能否产生立竿见影的效果？

带着这些问题，《商业价值》记者在第一时间专访了云适配创始人兼CEO、W3C中国区HTML5布道官陈本峰。以下为陈本峰对HTML5标准制定完成事件的解读提炼：

1.耗时8年 HTML5终完工，提前了六年

HTML5作为一个互联网标准，影响范围广泛，几乎所有人都会被影响，所以每向前推进一步都很谨慎。其实，HTML5标准预定2020年完工，显然这对于从业人员来说难以接受，也不符合互联网思维。后来，HTML5标准制定遵循快速迭代原则，让开发者能够第一时间享受到HTML5的最新功能。后续几年里，依然会有HTML5.1以及HTML5.2陆续放出。

2.HTML5命运的裁定者，W3C

W3C的作用主要是协调多方意见，真正参与标准制定还是W3C的会员，诸如微软、谷歌、苹果。这些会员拥有提案权，W3C会把这些公司提案的初稿放 到网站上，听取全球多方的意见，达成一致后才会定稿。当然，想要达成意见的一致并非易事，各家都会有自己的考虑。这就是W3C存在的意义，也是导致标准制 定耗时较长的原因之一。

3.几家欢喜几家愁

Kendo UI在2013年进行的全球开发者调查显示，HTML5已成为最受欢迎的跨平台应用开发工具。而标准完工对于开发者来说是振奋人心的，他们将在不久的将来真正实现很多HTML5的酷炫功能。此外，也为浏览器厂商指明了道路，他们终于可以大刀阔斧的执行HTML5标准了。此前他们在HTML5标准的的支持方面有些迟疑，因为标准的不确定，浏览器厂商会犹豫到底要不要完全遵循HTML5，一旦标准发生变化，浏览器厂商也需要做出相应的修改。

然而，该消息并非皆大欢喜。HTML5标准的完工，无疑给Web App增添了制胜的筹码。这对于 Native App从业者来说并不是一个好消息，同时受到影响的还有Native App的周边行业，如App分发、App数据统计公司。与App相比，HTML5有很强的渗透率，微信朋友圈本身就是HTML5网页，能够寄生在App 里。Web App体验不佳一直饱受诟病，其中一个原因就是浏览器性能支持不完整，相信标准完工后这一问题将会得到改善。

4.标准完工 并不是句号

该消息放出后，陈本峰在朋友圈写到，“Native App和Web App之争不久就会有个明朗的答案。”显然，标准完工并不会产生立竿见影的效果。

一方面，从一个技术流派迁移到另一个技术流派，需要一定时间。另一方面，HTML5是一个很大的概念，包含了很多功能，需要逐步实现。这个过程并非 是0到1，而是从0到100的过程。比如说HTML5有1万个功能，很多应用不一定需要1万个功能，也许1000个就足够了。但对有的应用来说远远不够， 这就需要浏览器将剩下的功能全部实现，开发者才能开发出越来越强大的应用。

原文链接：http://www.tmtpost.com/165028.html

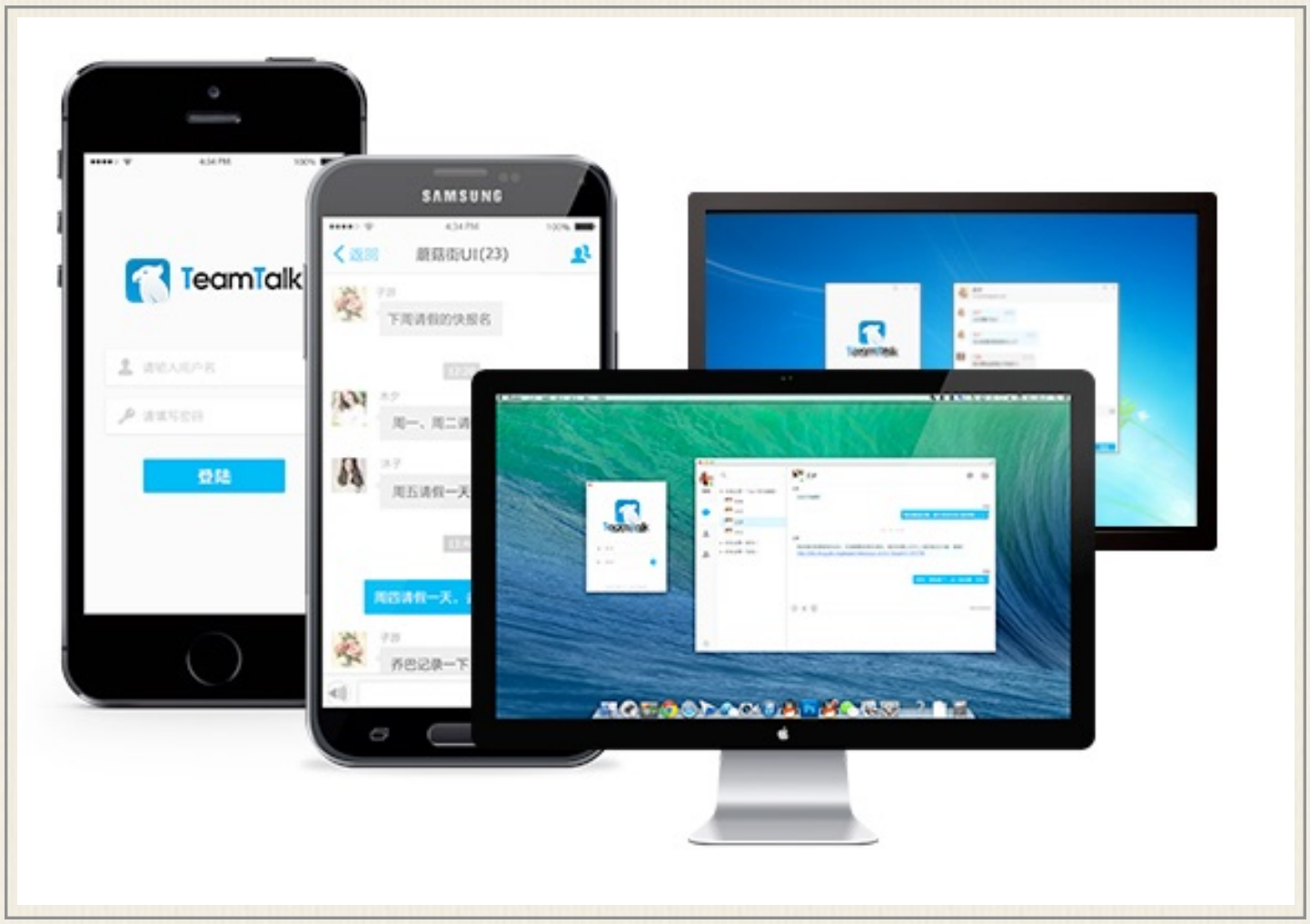
牛逼闪闪之蘑菇街开源 IM 系统 —— TeamTalk

作者：张远浩

项目背景

蘑菇街能有今天的快速发展，得益于开源软件群雄崛起的大环境背景，我们一直对开源社区怀有感恩之情，因此也一直希望能为开源社区贡献一份力量。

2013年我们蘑菇街从社区导购华丽转身时尚电商平台，为解决千万妹子和时尚卖家的沟通问题，我们开发了自己的即时通讯软件。既然已经有了用户使用 的IM，为什么我们自己公司内部沟通还要用第三方的呢？因此就有了TT(TeamTalk)的雏形，现在蘑菇街内部的在线沟通全部通过TT来完成。随着 TT功能的逐渐完善，我们决定把TT开源来回馈开源社区，希望国内的中小企业都能用上开源、免费、好用的IM工具！

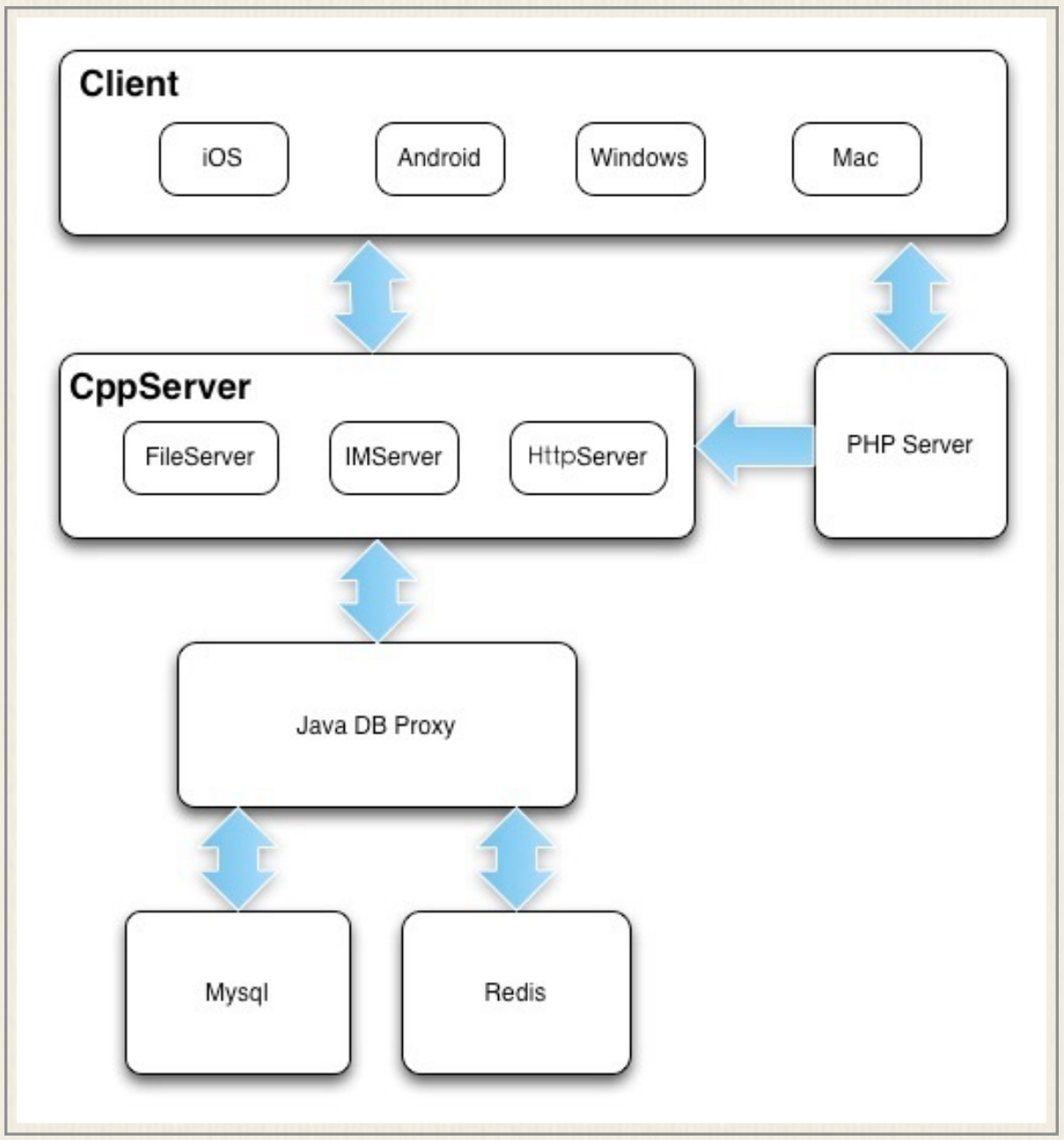


项目介绍

- 名称：TeamTalk
- 官网：http://tt.mogu.io/
- 开源协议：Apache License, Version 2.0
- 定位：中小型企业用户，member >= 2
- 特点：开源与产品并重
- 功能：可靠的消息传递机制；支持文字、图片、语音等富文本信息；文件收发等

项目框架

麻雀虽小五脏俱全，本项目涉及到多个平台、多种语言，简单关系如下图：



服务端：

CppServer：TTCppServer工程，包括IM消息服务器、http服务器、文件传输服务器、文件存储服务器、登陆服务器 java DB Proxy：TTJavaServer工程，承载着后台消息存储、redis等接口 PHP server：TTPhpServer工程，teamtalk后台配置页面

客户端：

- mac：TTMacClient工程，mac客户端工程
- iOS：TTIOSClient工程，IOS客户端工程
- Android：TTAndroidClient工程，android客户端工程
- Windows：TTWinClient工程，windows客户端工程
- 语言：c++、objective-c、java、php
- 系统环境：Linux、Windows，Mac, iOS, Android

原文链接：http://www.oschina.net/p/teamtalk

开源移动通讯架构与XMPP

作者：Tim

XMPP由于上下游良好的开源生态得到了广泛的采纳与应用，但是到了移动为主的时代，XMPP的不足也暴露出来。

XMPP全称是Extensible Messaging and Presence Protocol(也称为Jabber)，是一种支持消息及状态的协议，但在线状态在移动场景并是一个必需的feature。由于智能手机具有随时在线的 特点，状态可以视为永远在线；即便在app没有打开的情况下，系统可以通过push等方式发送最新的信息，因此大部分面向移动的通讯软件直接去掉了状态的 特性。因此设计成支持多终端状态的XMPP在移动领域并不是擅长之地。另外一方面XMPP是一种基于XML的协议，它的请求及应答机制也是主要为稳定长连 网络环境所设计，对于带宽偏窄及长连不稳定的移动网络并不是特别优化，因此它的弊端就充分暴露出来了。

移动领域也有不少非XMPP的开源实现，尽管赢得主流认可的不多。比较商业化的有telegram，其特性跟微信比较类似，在一些国家也取得了飞速发展，尽管其宣称的开源还不完全。国内有前几天蘑菇街发布的的TeamTalk , 其开源地址是 https://github.com/mogutt 从发布的短短几天来看， star/fork 数非常可观。其介绍支持企业内部通讯场景，且具有完整的移动端支持。

最近也在考虑在这方面做一些尝试，一个理想的适合移动时代的IM开源软件，它应该具有哪些因素?

- 分布式扩展能力。在XMPP领域，由于Openfire的简单易用，成为很多团队首选的方案，但使用Openfire的团队都需要接着思考扩展openfire的分布式扩展能力，以便承担更大的用户访问规模。

- 移动友好的协议，协议具有良好的长连及短连自适应能力，具有数据的增量更新能力，较低的重连成本等。有网友推荐MQTT http://mqtt.org/，尚未深入评估。

- 移动SDK，主要实现协议层及网络逻辑，以简化客户端接入及开发成本。

- 开发语言，更多考虑一些能带来编程乐趣的新型语言，在一定程度上，scale的问题可以做到与语言无关。

在微博上交流的时候，一些网友还提到了XMPP的federation功能，federation类似邮件协议，利用XMPP可以实现多个域的用户 互联互通。但由于此功能对于企业的商业化利益较难看清，因此在过去很长一段时间都没有得到充分发展。Google最近也放弃了多年来坚持的XMPP federation的支持。

需要补充的是，尽管前面提到XMPP种种问题，但是也别低估一种新协议的认知成本，很多时候选择XMPP并不是因为它协议强大或多么适合用户的场 景，而是当大众群体已经了解一种协议之后，即使这种方式存在种种问题，但还是较难广泛认可及接受一种方式或协议。在没有特殊原因的情况下，普通的通讯场景 仍然建议使用XMPP方式。

原文链接：http://timyang.net/im/mobile-im-xmpp/

使用C/C++ 扩展Python

作者：麦田守望

前期的网页抽取算法使用C++开发，为了提升代码复用，减少维护成本，项目中决定封装成Python扩展方便Python使用。
Python与C/C++互操作有很多方案：Python C API, swig, sip, ctypes, cpython, cffi, boost.python等。这里选择了最原始的Python C API方式。

一、开发前准备

1.Python对象

大 多数Python对象在Python解析器中都为PyObject，在C代码中只能声明PyObject*类型的python对象，然后使用该对象对应的 初始化函数初始化。如PyTuple_New,PyList_New,PyDict_New,Py_BuildValue等。
例如构建一个{‘a':{‘b':['123','34']}}对象

```
PyObject* obj = PyDict_New();

PyObject* b = PyDict_New();

PyObject* c = PyList_New(2);

PyList_SetItem(c, 0, Py_BuildValue("s", "123"));

PyList_SetItem(c, 1, Py_BuildValue("s", "34"));

PyDict_SetItem(a, "b", c);

PyDict_SetItem(obj, "a", a);
```

Python对象问题这里有一些文档：

http://docs.python.org/2/c-api/intro.html#objects-types-and-reference-counts

http://docs.python.org/2/c-api/dict.html

http://docs.python.org/2/c-api/list.html

2.Python内存管理

Python 对象管理采用引用技术模型，内部有一些复杂的循环引用等处理措施。主要有 Py_INCREF() / Py_DECREF()两个宏负责处理。具体文档可以看这里http://docs.python.org/2/c-api /intro.html#reference-counts

例如上一点申请的对象obj如果需要释放怎么办？不可以直接free/delete，直接Py_DECREF(obj)，然后obj = NULL即可，否则会报错。

3.线程安全

Python 由于历史比较悠久，作者在开发的时候可能并没有考虑到多线程这个东西，因为Python的内存管理并不是线程安全的。在后来后来版本中为了处理这个线程安 全问题引入了GIL即global interpreter lock。这是一个粗粒度的锁，执行Python ByteCode之前都会取得这个锁。以至于Python的多线程比较鸡肋，GIL也就成了性能瓶颈。这个问题很多地方都有讨论，我之前有一篇文章专门对 这个问题进行了说明，感兴趣的同学请去这里http://in.sdo.com/?p=1623。

有人会问为什么不设计更细粒度的锁?实际上有人已经进行了尝试,但是为了不增加实现的复杂性也就一直没有加到CPython中。其他版本的python如IronPython等对这个问题已经做了改善。

实际开发时有两种情况需要关心：

1).释放锁

这种情景只要在进行IO或CPU繁重的计算时，暂时释放GIL使得其他线程的代码可以执行。

2).取得锁

主要出现在C回调Python代码

参考文档:

http://docs.python.org/2/c-api/init.html#thread-state-and-the-global-interpreter-lock

二、开发扩展

有了上面的知识我们开始进行实际的开发。

1.导出函数

写好C API函数之后我们需要导出，写一个函数描述表即可，如下面的EchoMethods，一定要以NULL结尾。

```
PyObject* echo(PyObject* self, PyObject* args)
{
    char* input = NULL;
if(!PyArg_ParseTuple(args, "s", &input))
{
    printf("parse arg errorn");
    return NULL;
}

int count = 0;
do
{
    printf("%sn", input);
    count++;
```

```
        }while(count &lt; 100);
        return Py_BuildValue("i", 0);
}

static PyMethodDef EchoMethods[] =
{
        {"echo", (PyCFunction)echo, METH_VARARGS},
        {NULL, NULL}
};
```

2.导出对象

除了上面提到的使用复杂的PyObject操作语法封装一个Python对象返回之外还有其他途径，如直接导出C的Struct到Python。这里不详谈，需要的可以查相关资料。

3.初始化模块

模块初始化调用Py_InitModule，传入模块名和模块的方法描述表即可。如果初始化失败会返回error可以做相应处理。

```
PyMODINIT_FUNC initecho()
{
        Py_InitModule("echo", EchoMethods);
}
```

三、编译与使用

1.如何编译、分发、使用

上面这些代码当然会用到python-devel库。编译的时候使用GCC直接编译成一般的so，就可以直接在python里面调用了。Python会自己选择如何加载这个so。

g++ -c echo.c -I /usr/include/python2.7/include/python2.7 -fPIC

g++ -shared echo.o -o echo.so

上面已经提到了，实际上把自己编译好的so放在PYTHONPATH路径中的任意一个下面都可以直接调用了。

2.更便捷的方式

上面的编译方式可以自己写一个Makefile处理起来更灵活，实际上Python有一个更方便的处理方式。使用distutils包，编译安装一步到位，这也是easy_install等工具使用的方式。
上面这个简单使用distutils处理起来像这样:

```
from distutils.core import setup, Extension
echomodule = Extension("echo",
                       sources = ["echo.c"])
setup(name = "echo",
      version = "1.0",
      description = "test",
      author = "dudu"
      ext_modules = [echomodule])
```

Extension对象定义一个扩展的源文件、需要用到的第三方库、头文件、特殊的编译选项等等，而setup则定义安装的规则及扩展的一些属性。

使用的时候执行下面两个命令就可以了。

python setup.py build

sudo python setup.py install

这部分可以参考http://docs.python.org/2/distutils/apiref.html

文章是写完了。特别推荐需要开发许多接口的人去看看开头提到的swig/sip等等，这些项目只需要编写简单的规则，就可以为c/c++中的方法生成wrapper。这里只所以有采用c api是因为需求简单，需要暴露给python的总共也没几个函数。

原文链接：http://www.the520.cn/2014/02/27/python_c_api_extension.htm

JVM必备指南

译者：xiafei

简介

Java虚拟机（JVM）是Java应用的运行环境，从一般意义上来讲，JVM是通过规范来定义的一个虚拟的计算机，被设计用来解释执行从Java 源码编译而来的字节码。更通俗地说，JVM是指对这个规范的具体实现。这种实现基于严格的指令集和全面的内存模型。另外，JVM也通常被形容为对软件运行 时环境的实现。通常JVM实现主要指的是HotSpot。

JVM规范保证任何的实现都能够以同样的方式解释执行字节码。其实现可以多样化，包括进程、独立的Java操作系统或者直接执行字节码的处理器芯 片。我们了解最多的JVM是作为软件实现，运行在流行的操作系统平台上（包括Windows、OS X、Linux和Solaris等）。

JVM的结构允许对一个Java应用进行更细微的控制。这些应用运行在沙箱（Sandbox）环境中。确保在没有恰当的许可时，无法访问到本地文件系统、处理器和网络连接。远程执行时，代码还需要进行证书认证。

除了解释执行Java字节码，大多数的JVM实现还包含一个JIT（just-in-time 即时）编译器，用于为常用的方法生成机器码。机器码使用的是CPU的本地语言，相比字节码有着更快的运行速度。

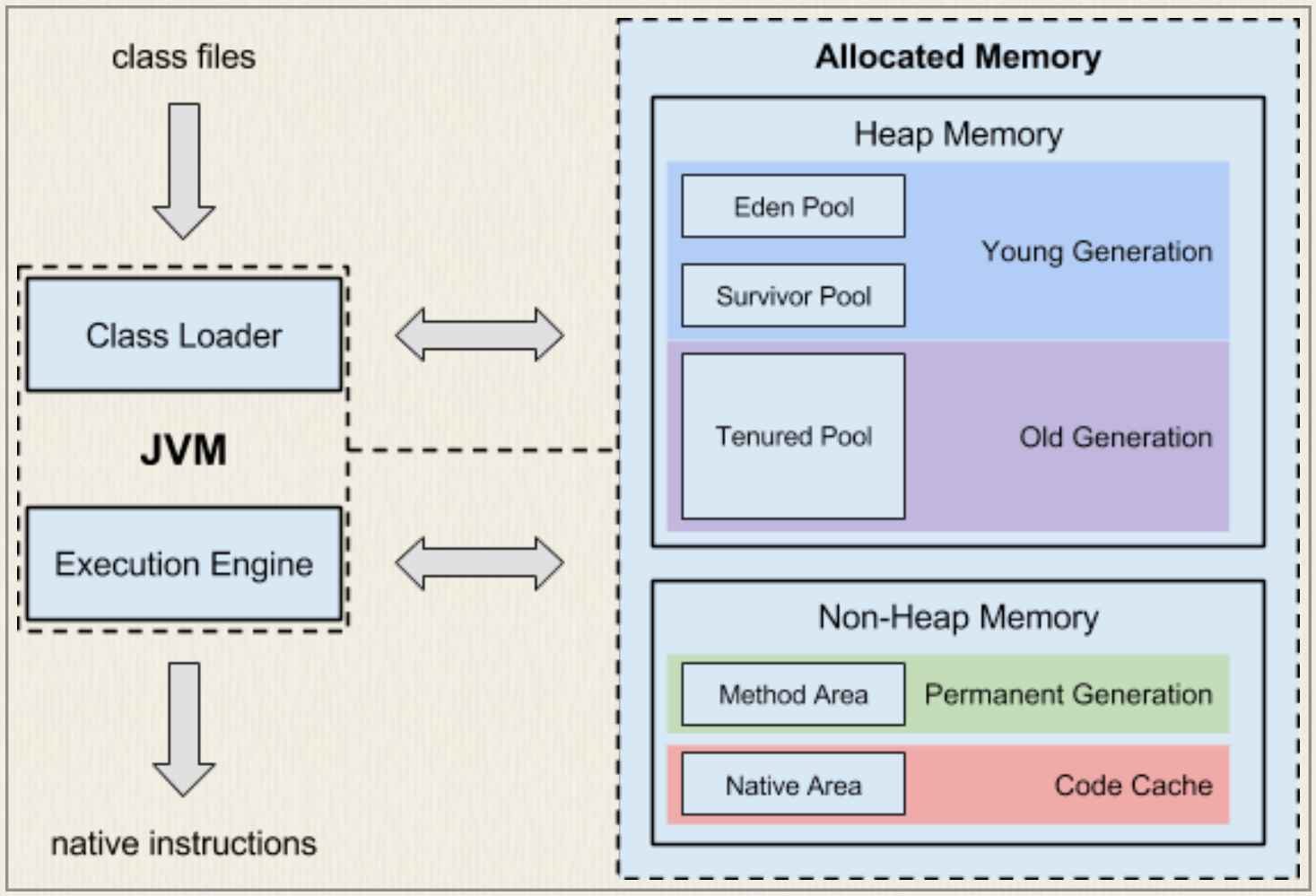
虽然理解JVM不是开发或运行Java程序的必要条件，但是如果多了解一些JVM知识，那么就有机会避免很多性能上的问题。理解了JVM，实际上这些问题会变得简单明了。

体系结构

JVM规范定义了一系列子系统以及它们的外部行为。JVM主要有以下子系统:

- Class Loader 类加载器。 用于读入Java源代码并将类加载到数据区。
- Execution Engine 执行引擎。 执行来自数据区的指令。

数据区使用的是底层操作系统分配给JVM的内存。



类加载器（Class Loader）

JVM在下面几种不同的层面使用不同的类加载器:

- bootstrap class loader（引导类加载器）：是其他类加载器的父类，它用于加载Java核心库，并且是唯一一个用本地代码编写的类加载器。

- extension class loader（扩展类加载器）：是bootstrap class loader加载器的子类，用于加载扩展库。
- system class loader（系统类加载器）：是extension class loader加载器的子类，用于加载在classpath中的应用程序的类文件。
- user-defined class loader（用户定义的类加载器）：是系统类加载器或其他用户定义的类加载器的子类。

当一个类加载器收到一个加载类的请求，首先它会检查缓存，确认该类是否已经被加载，然后把请求代理给它的父类。如果父类没能成功的加载类，那么子类 就会自己去尝试加载该类。子类可检查父类加载器的缓存，但父类不能看到子类所加载的类。之所类加载体系会这样设计，是认为一个子类不应该重复加载已经被父 类加载过的类。

执行引擎（Execution Engine）

执行引擎一个接一个地执行被加载到数据区的字节码。为了保证字节码指令对于机器来说是可读的，执行引擎使用下面两个方法：

- 解释执行：执行引擎把它遇到的每一条指令解释为机器语言。
- 即时编译：如果一条指令经常被使用，执行引擎会把它编译为本地代码并存储在缓存中。这样，所有和这个方法相关的代码都会直接执行，从而避免重复解释。

尽管即时编译比解释执行要占用更多的时间，但是对于需要使用成千上万次的方法，只需要处理一次。相比每次都解释执行，以本地代码的方式运行会节约很多执行时间。

JVM规范中并不规定一定要使用即时编译。即时编译也不是用于提高JVM性能的唯一的手段。规范仅仅规定了每条字节码对应的本地代码，至于执行引擎如何实现这一对应过程的，完全由JVM的具体实现来决定。

内存模型（Memory Model）

Java内存模型建立在自动内存管理的概念之上。当一个对象不再被一个应用所引用，垃圾回收器就会回收它，从而释放相应的内存。这一点和其他很多需要自行释放内存的语言有很大不同。

JVM从底层操作系统中分配内存，并将它们分为以下几个区域：

- 堆空间（Heap Space）：这是共享的内存区域，用于存储可以被垃圾回收器回收的对象。
- 方法区（Method Area）：这块区域以前被称作“永生代”（permanent generation），用于存储被加载的类。这块区域最近被JVM取消了。现在，被加载的类作为元数据加载到底层操作系统的本地内存区。
- 本地区（Native Area）：这个区域用于存储基本类型的引用和变量。

一个有效的管理内存方法是把对空间划分为不同代，这样垃圾回收器就不用扫描整个堆区。大多数的对象的生命周期都很段短暂，那些生命周期较长的对象往往直到应用退出才需要被清除。

当一个Java应用创建了一个对象，这个对象是被存储到“初生池”（eden pool）。一旦初生池存储满了，就会在 新生代触发一次minor gc（小范围的垃圾回收）。首先，垃圾回收器会标记出那些“死对象”（不再被应用所引用的对象），同时延长所有保留对象的生命周期（这个生命周期长度是用 数字来描述，代表了期所经历过的垃圾回收的次数）。然后，垃圾回收器会回收这些死对象，并把剩余的活着的对象移动到“幸存池”（survivor pool），从而清空初生池。

当一个对象存活达到一定的周期后，它就会被移动到堆中的老生代：“终身代”（tenured pool）。最后，当终身代被填满时，就会触发一次full gc或major gc（完全的垃圾回收），以清理终身代。

（译者注：一般我们把初生池和幸存池所在的区域合并成为新生代，把终身代所在的区域成为老生代。对应的，在新生代上产生的gc称为minor gc，在老生代上产生的gc称为full gc。希望这样大家在其他地方看到对应的术语时能更好理解）

当垃圾回收（gc）执行的时候，所有应用线程都要被停止，系统产生一次暂停。minor gc非常频繁，所以被优化的能够快速的回收死对象，是新生代的内存的主要的回收方式。major gc运行起来就相对慢得多，因为要扫描非常多的活着的对象。垃圾回收器本身也有多种实现，有些垃圾回收器在一定情况下能更快的执行major gc。

堆的大小是动态的，只有堆需要扩张的时候才会从内存中分配。当堆被填满时，JVM会重新给堆分配更多的内存，直到达到堆大小的上限，这种重新分配同样会导致应用的短暂停止。

线程

JVM是运行在一个独立的进程中的，但它可以并发执行多个线程，每个线程都运行自己的方法，这是Java必备的一个部分。以即时消息客户端这样一个 应用为例，它至少运行两个线程。一个线程用于等待用户输入，另一个检查服务端是否有新的消息传输。再以服务端应用为例，有时一个请求可能要涉及多个线程并 发执行，所以需要多线程来处理请求。

在JVM的进程中，所有的线程共享内存和其他可用的资源。每一个JVM进程在进入点（main方法）处都要启动一个主线程，其他线程都从主线程启 动，成为执行过程中的一个独立部分。线程可以再不同的处理器上并行执行，同样也可以共享一个处理器，线程调度器负责处理多个线程共享一个处理器的情况。

很多应用（特别是服务端应用）会处理很多任务，需要并行运行。这些任务中有些是非常重要的，需要实时执行的。而另外一些是后台任务，可以在CPU空 闲时执行。任务是在不同的线程中运行的。举例子来说，服务端可能有一些低优先级的线程，它们会根据一些数据来计算统计信息。同时也会启动一些高优先级的进 程用于处理传入的数据，响应对这些统计信息的请求。这里可能有很多的源数据，很多来自客户端的数据请求，每个请求都会使服务端短暂的停止后台计算的线程以 响应这个请求。所以，你必须监控在运行的线程数目并且保证有足够的CPU时间来执行必要的计算。

（译者注：这一段在原文中是在性能优化的章节，译者认为这可能是作者的不小心，似乎放在线程的章节更合适。）

性能优化

JVM的性能取决于其配置是否与应用的功能相匹配。尽管垃圾回收器和内存回收进程是自动管理内存的，但是你必须掌管它们的频率。通常来说，你的应用可使用的内存越多，那么这些会导致应用暂停的内存管理进程需要起作用的就越少。

如果垃圾回收发生的频率比你想的要多很多，那么可以在启动JVM的时候为其配置更大的最大堆大小值。堆被填满的时间越久，就越能降低垃圾回收发生的频率。最大堆大小值可以在启动JVM的时候，用-Xmx参数来设定。默认的最大堆大小是被设置为可用的操作系统内存的四分之一，或者最小1GB。

如果问题出在经常重新分配内存，那么你可以把初始化堆大小设置为和最大堆大小一样。这就意味着JVM永远不需要为堆重新分配内存。但这样做就会失去动态堆大小适配的优化，堆的大小从一开始就被固定下来。配置初始化对大小是在启动JVM，用-Xms来设定。默认初始化堆大小会被设定为操作系统可用的物理内存的六十四分之一，或者设置一个最小值。这个值是根据不同的平台来确定的。

如果你清楚是哪种垃圾回收（minor gc或major gc）导致了性能问题，可以在不改变整个堆大小的情况下设定新生代和老生代的大小比例。对于需要产生大量临时对象的应用，需要增大新生代的比例（当然，后 果是减小了老生代的大小）。对于长生命周期对象较多的应用，则需增大老生代的比例（自然需要减少新生代的大小）。以下几种方法可以用来设定新生代和老生代 的大小：

- 在启动JVM时，使用-XX:NewRatio参数来具体指定新生代和老生代的大小比例。比如，如果想让老生代的大小是新生代的五倍，则设置参数为-XX:NewRatio=5，默认这个参数设定为2（即老生代占用堆空间的三分之二，新生代占用三分之一）。
- 在启动JVM时，直接使用-Xmn参数设定初始化和最大新生代大小，那么堆中的剩余大小即是老生代的大小。

- 在启动JVM时，直接使用-XX:NewSize和-XX:MaxNewSize参数设定初始化和最大新生代大小，那么堆中的剩余大小即是老生代的大小。

每一个线程都有一个栈，用于保存函数调用、返回地址等等，这些栈有着对应的内存分配。如果线程过多，就会导致OutOfMemory错误。即使你有 足够的空间的堆来存放对象，你的应用也可能会因为创建一个新的线程而崩溃。这种情况下，需要考虑限制线程中的栈大小的最大值。线程栈大小可以在JVM启动 的时候，通过-Xss参数来设置，默认这个值被设定为320KB至1024KB之间，这和平台相关。

性能监控

当开发或运行一个Java应用的时候，对JVM的性能进行监控是很重要的。配置JVM不是一次配置就万事大吉的，特别是你要应对的是Java服务器 应用的情况。你必须持续的检查堆内存和非堆内存的分配和使用情况，线程数的创建情况和内存中加载的类的数据情况等。这些都是核心参数。

使用Anturis控制台，你可以为任何的硬件组件上运行的JVM配置监控（例如，在一台电脑上运行的一个Tomcat网页服务器）。

JVM监控可以使用以下衡量标准:

- 总内存使用情况（MB）：即JVM使用的总内存。如果JVM使用了所有可用内存，这项指标可以衡量底层操作系统的整体性能。

- 堆内存使用（MB）：即JVM为运行的Java应用所使用的对象分配的所有内存。不使用的对象通常会被垃圾回收器从堆中移除。所以，如果这个指数增大，表示你的应用没有把不使用的对象移除或者你需要更好的配置垃圾回收器的参数。

- 非堆内存的使用（MB）：即为方法区和代码缓存分配的所有内存。方法区是用于存储被加载的类的引用，如果这些引用没有被适当的清理，永生代池会在每次应用被重新部署的时候都会增大，导致非堆的内存泄露。这个指标也可能指示了线程创建的泄露。

- 池内总内存（MB）：即JVM所分配的所有变量内存池的内存和（即除了代码缓存区外的所有内存和）。这个指标能够让你明确你的应用在JVM过载前所能使用的总内存。
- 线程：即所有有效线程数。举个例子，在Tomcat服务器中每个请求都是一个独立的线程来处理，所以这个衡量指标可以表示当前有多少个请求数，是否影响到了后台低权限的线程的运行。
- 类：即所有被加载的类的总数。如果你的应用动态的创建很多类，这可能是服务器内存泄露的一个原因。

译文链接：http://www.importnew.com/13556.html

原文链接：https://anturis.com/blog/java-virtual-machine-the-essential-guide/

编写最简单的内核：HelloWorld

译者：NOALFGO

内核是操作系统最核心的内容，主要提供硬件抽象层、磁盘及文件系统控制、多任务等功能，由于其涉及非常广泛的计算机知识，很少被人们所熟悉，因而披上了一层神秘的面纱。
本文将从零开始实现一个最简单的内核，其可以通过x86系统的GRUB引导启动，并向屏幕输出“Hello World!“字符串。该内核代码非常简短，并且在本人的Debian 7系统中可以正常运行。

x86机器启动过程

在具体实现这个内核之前，我们先看看机器具体是怎么启动并且把控制权交给内核的。

x86的CPU固定地在物理地址为[0xFFFFFFF0]的地方开始运行，这是32位地址空间的最后16个字节。这里只包含了一个跳转指令，跳转到BIOS把它自己拷贝到的内存区域的地址。

然后，BIOS开始执行。它首先根据配置的设备启动顺序依次寻找可启动的设备（根据一个特定的魔数可以决定一个设备是否启动）。一旦找到一个可启动的设 备，它就把该设备第一个扇区的内容复制到RAM中物理地址从[0x7C00]开始的地方，然后跳转到该地址并且开始执行那里加载的代码。这段代码称为启动 引导装载程序(bootloader)。Bootloader然后在物理地址为[0x100000]的地方加载内核，地址[0x100000]就是x86 机器上内核的起始地址。

需要的工具

- 一台x86电脑
- Linux
- NASM汇编器
- gcc
- ld（GNU链接器）
- grub

汇编入口点

我们希望用C来写所有的代码，但免不了要写一点汇编代码。我们会写一个x86汇编语言的小文件来作为内核的起始点，这段汇编所做的事情就是调用一个我们用C写的外部函数，然后停止程序运行。

怎么确定这段汇编代码会作为内核的起始点呢?

我们会使用一个链接脚本来链接所有的目标文件来产生一个最终的内核可执行映像。在这个链接脚本中，我们会显式指明二进制文件要加载在地址为[0x100000]的地方，这就是内核所在的地方。于是，bootloader会负责触发这个内核的入口点。

以下是汇编代码:

```
;;kernel.asm，内核汇编代码
bits 32          ;nasm伪指令
section .text   ;代码段

global start    ;全局变量
extern kmain    ;kmain定义在C文件中
```

```
start:
    cli         ;禁止中断
    call kmain  ;调用kmain函数
    hlt         ;终止CPU运行
```

第一条指令中的bit 32不是x86汇编指令，而是NASM汇编器的伪指令，表明将会产生一段运行在32位处理器上代码。这句代码不是必须的，但显示加上会是一个好的编程实践。

第二行开始就是代码段，即放置代码的地方。
global也是NASM的伪指令，表示把源代码中的一个符号设置成全局符号。于是链接器知道start符号在哪里，其实这就是我们的入口点。
kmain是将会在kernel.c中实现的一个函数，extern表明这个函数会在其他地方定义。

于是，我们有了start函数，它会调用kmain函数，然后通过hlt指令停止CPU。由于中断会从hlt指令中唤醒CPU，所以我们事先使用cli（意为clear interrupts）指令禁止中断。

C语言内核

我们在kernel.asm中调用kmain()函数，所以C代码会从kmain()开始执行。

```
//kernel.c文件
void kmain(void)
{
  char *str = "Hello World！";
```

```
char *vidptr = (char*)0xb8000; //显存开始地址
unsigned int i = 0;
unsigned int j = 0;
//清空屏幕，共25行，每行80个字符，每个字符2字节
while(j < 80 * 25 * 2) {
        //空白字符
        vidptr[j] = ' ';
        //属性字节：黑色背景，灰色前景
        vidptr[j+1] = 0x07;
        j = j + 2;
}
j = 0;
    while(str[j] != '\0') {
        vidptr[i] = str[j];
        vidptr[i+1] = 0x07;
        ++j;
        i = i + 2;
    }
    return;
}
```

这里内核所做的事情就是：清空屏幕，打印字符串“Hello World!”。

首先是指针vidptr指向地址[0xb8000]，这是保护模式下显存的开始地址。屏幕的文本内存只是地址空间的一连串内存区域，它从 [0xb8000]开始映射屏幕的输入输出，支持25行，每行80个ASCII字符，每个字符用16位

（2字节）表示，而不是我们熟悉的8位（1字节）。 2字节中第1个字节是该字符的ASCII表示，第2个字节是属性字节，描述字符的包括颜色在内的属性。如果要让背景为黑色而字体为绿色，可以在第1个字节 保存字符的ASCII值，在第2个字节保存值[0x02]：0代表黑色背景，2代表绿色前景。

其它颜色属性定义如下：

0	1	2	3	4	5	6	7
Black	Blue	Green	Cyan	Red	Magenta	Brown	Light Grey
8	9	10	11	12	13	14	15
Dark Grey	Light Blue	Light Green	Light Cyan	Light Red	Light Magenta	Light Brown	White

我们的内核使用了黑色背景以及灰色字体，所以属性字节为[0x07]。

在第一个while循环中，程序在所有的25行80列中写入空字符和[0x07]属性，从而清空了屏幕。
在第二个while循环中，字符串”Hello World!“被写到了显存的开始区域，每个字符仍是拥有[0x07]属性。这就在屏幕上打印了该字符串。

链接部分

使用NASM把kernel.asm编译成目标文件，再使用GCC把kernel.c编译成另一个目标文件，然后需要把这两个目标文件链接成一个可以启动的内核映像。
我们使用链接脚本来达到这个目的，链接脚本可以作为参数传递进链接器ld中以控制链接的过程。

*//link.ld*文件

OUTPUT_FORMAT(elf32-i386)

```
ENTRY(start)
SECTIONS
{
    . = 0x100000;
    .text : { *(.text) }
    .data : { *(.data) }
    .bss  : { *(.bss)  }
}
```

OUTPUT_FORMAT设置输出的可执行文件为32位的ELF文件，ELF是x86架构上类Unix系统的标准二进制文件格式。

ENTRY接受一个参数，指定其为最终可执行文件的入口点。

SECTION是这里最关键的部分，它指定不同的段怎么合并以及放在什么地方，从而定义最终可执行文件的布局。
大括号内就是SECTION的语句，句点(.)为位置计数器，一般被初始化为SECTIONS块开始的地方[0x0]，但可以任意修改。因为内核代码需要在地址[0x100000]处开始，所以设置位置计数器为[0x100000]。
第二行中的星号是通配符，可以匹配任何文件名，*(.text)即表示匹配所有输入文件的代码段。于是，链接器合并所有目标文件的代码段到可执行文件的代 码段中，具体地址由位置计数器决定，这里即为[0x100000]。链接器产生代码段后，位置计数器会变成：0×100000 + 输出代码段的大小。
同样地，数据段和bss段会被合并，并放置在位置计数器指定的地方。

Grub和多重引导

现在已经准备好了构建内核的所有文件了，但要用GRUB进行引导还需要最后一个步骤。

多重引导规范(Multiboot specification)是一个使用bootloader加载不同X86内核的标准，GRUB只会加载满足这个规范的内核。根据这个规范，内核必须在 它的前8KB字节中包含头信息（Multiboot header）。这个头信息包含4字节对齐的3个域，分别为：

- 魔数域：包含魔数[0x1BADB002]。
- 标志域：这里不关心这个域，置为0。
- 校验和域：检验和域和前面两个域相加之后的结果必须为0。

于是kernel.asm应该修改为，代码中的dd表示定义一个4字节的双字：

```
;;kernel.asm，内核汇编代码

bits 32         ;nasm伪指令
section .text   ;代码段
        ;多重引导规范
        align 4
        dd 0x1BADB002             ;魔数
        dd 0x00                   ;标志
        dd - (0x1BADB002 + 0x00) ;校验和

global start    ;全局变量
extern kmain    ;kmain定义在C文件中

start:
    cli         ;禁止中断
    call kmain  ;调用kmain函数
```

```
    hlt         ;终止CPU运行
```

构建内核

现在可以从kernel.asm和kernel.c生成目标文件，然后使用连接脚本进行链接。

使用汇编器nasm产生ELF-32格式的目标文件kasm.o：

nasm -f elf32 kernel.asm -o kasm.o

使用编译器gcc产生目标文件kc.o，"-c"参数保证只编译，不进行链接：

gcc -m32 -c kernel.c -o kc.o

使用链接器ld根据链接控制脚本产生可执行映像文件kernel：

ld -m elf_i386 -T link.ld -o kernel kasm.o kc.o

配置GRUB并运行内核

GRUB需要内核以kernel-<version>形式命名，于是把内核kernel重命名为kernel-701，并利用超级管理员权限放到/boot目录下。

对于bootloader为GRUB的发行版，修改配置文件/boot/grub/grub.cfg，添加以下条目：

```
title myKernel
  root (hd0,0)
  kernel /boot/kernel-701 ro
```

对于*bootloader*为*GRUB2*的发行版，添加的配置应该为：

```
menuentry 'kernel 701' {
  set root='hd0,msdos1'
  multiboot /boot/kernel-701 ro
```

}

重启电脑，选择GRUB列表中新增加的kernel-701内核选项，这时可以看到屏幕上显示”Hello World!“。
这就是你的内核！

译文链接：http://noalgo.info/649.html

原文链接：http://arjunsreedharan.org/post/82710718100/kernel-101-lets-write-a-kernel

Swift 命名空间

作者：王巍

Objective-C 一个一直以来令人诟病的地方就是没有命名空间，在应用开发时，所有的代码和引用的静态库最终都会被编译到同一个域和二进制中。这样的后果是一旦我们有重复 的类名的话，就会导致编译时的冲突和失败。为了避免这种事情的发生，Objective-C 的类型一般都会加上两到三个字母的前缀，比如 Apple 保留的 NS 和 UI 前缀，各个系统框架的前缀 SK (StoreKit)，CG (CoreGraphic) 等。Objective-C 社区的大部分开发者也遵守了这个约定，一般都会将自己名字缩写作为前缀，把类库命名为 AFNetworking 或者 MBProgressHUD 这样。这种做法可以解决部分问题，至少我们在直接引用不同人的库时冲突的概率大大降低了，但是前缀并不意味着不会冲突，有时候我们确实还是会遇到即使使用 前缀也仍然相同的情况。另外一种情况是可能你想使用的两个不同的库，分别在它们里面引用了另一个相同的很流行的第三方库，而又没有更改名字。在你分别使用 这两个库中的一个时是没有问题的，但是一旦你将这两个库同时加到你的项目中的话，这个大家共用的第三方库就会和自己发生冲突了。

在 Swift 中，由于可以使用命名空间了，即使是名字相同的类型，只要是来自不同的命名空间的话，都是可以和平共处的。和 C# 这样的显式在文件中指定命名空间的做法不同，Swift 的命名空间是基于 module 而不是在代码中显式地指明，每个 module 代表了 Swift 中的一个命名空间。也就是说，同一个 target 里的类型名称还是不能相同的。在我们进行 app 开发时，默认添加到 app 的主 target 的内容都是处于同一个命名空间中的，我们可以通过创建 Cocoa (Touch) Framework 的 target 的方法来新建一个 module，这样我们就可以在两个不同的 target 中添加同样名字的类型了：

```
// MyFramework.swift
```

```
// 这个文件存在于 MyFramework.framework 中
public class MyClass {
    public class func hello() {
        println("hello from framework")
    }
}

// MyApp.swift
// 这个文件存在于 app 的主 target 中
class MyClass {
    class func hello() {
        println("hello from app")
    }
}
```

在使用时，如果出现可能冲突的时候，我们需要在类型名称前面加上 module 的名字 (也就是 target 的名字)：

```
MyClass.hello()
// hello from app

MyFramework.MyClass.hello()
// hello from framework
```

因为是在 app 的 target 中调用的，所以第一个 MyClass 会直接使用 app 中的版本，第二个调用我们指定了 MyFramework 中的版本。

另一种策略是使用类型嵌套的方法来指定访问的范围。常见做法是将名字重复的类型定义到不同的 struct 中，以此避免冲突。这样在不使用多个 module 的情况下也能取得隔离同样名字的类型的效果：

```
struct MyClassContainer1 {
    class MyClass {
        class func hello() {
            println("hello from MyClassContainer1")
        }
    }
}

struct MyClassContainer2 {
    class MyClass {
        class func hello() {
            println("hello from MyClassContainer2")
        }
    }
}
```

使用时：

```
MyClassContainer1.MyClass.hello()
MyClassContainer2.MyClass.hello()
```

其实不管哪种方式都和传统意义上的命名空间有所不同，把它叫做命名空间，更多的是一种概念上的宣传。不过在实际使用中只要遵守这套规则的话，还是能避免很多不必要的麻烦的，至少唾手可得的是我们不再需要给类名加上各种奇怪的前缀了。

原文链接：http://swifter.tips/namespace/

美团性能分析框架和性能监控平台

作者：shijun

以下是我在 Velocity China 2014 做的题为“美团性能分析框架和性能监控平台”演讲的主要内容，现在以图文的形式分享给大家。

今天讲什么?

性能的重要性不言而喻，需要申明的是，我们今天不讲业界最佳性能实践，这些实践已经有很多沉淀，具体可以参考《高性能网站》和《高性能浏览器网络》 等书，另外，我们不打算讲性能优化的结果指标，比如页面完全加载时间，首屏时间，结果指标固然重要，是我们工作成果的量化衡量，但是对于做性能优化工作的 工程师来说，过程指标对其起到的帮助作用更大。

既然不讲最佳实践，那讲什么呢？我们按最佳实践提供的方法去实践，但是后来遇到了瓶颈，到底遇到了什么瓶颈？我们是如何突破这个瓶颈的？成效如何？这些对在座的各位又有什么借鉴意义呢?

遇到什么瓶颈?

在遇到瓶颈之前，我们做了很多工作，主要包括:

- 简单的数据采集，包括完全加载时间，DomReady 时间，需要注意的是这些都是结果指标;
- 依照“业界最佳实践”快糙猛的做了很多事情：比如异步化，静态化，LazyLoading，BigRender，这些实践效果都还不错;
- 因为只有结果指标数据，这个阶段我们绝大部分决策都是基于别人的经验，甚至拍脑袋，而不是基于应用的实际性能细节数据;

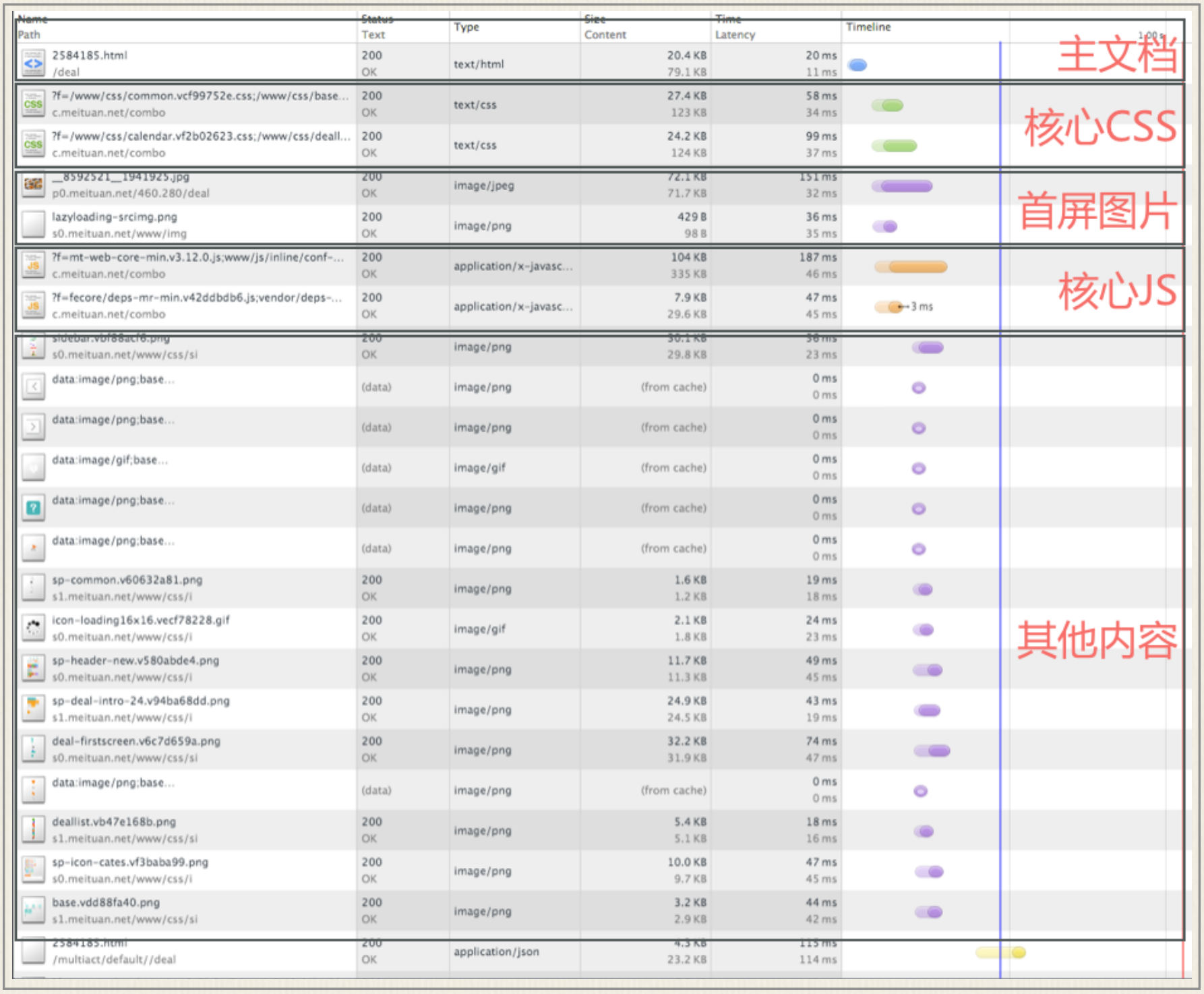
快糙猛的方式注定不是可持续的，很快，我们遇到了瓶颈，具体是什么瓶颈呢?

- 首先，如果把业界最佳实践当成燃料，而性能优化当成驾车远行的话，我们的燃料很快就烧完了，因为大家总结出来的通用的优化手段总是有限的，而我们的目标还没有达到;
- 其次，因为我们只采集了结果指标，只知道整体表现如何，面对异常波动我们显得特别无力，因为显示世界影响性能的因素太多了，对于到底发生什么事情了，我们无从得知;
- 再次，由于对性能缺少内窥，我们无法找到更多的优化点，实际上，我们需要一个类似于显微镜的东西，来看看应用内部还有哪些可优化的地方;

如何突破瓶颈?

面对这些瓶颈，我们需要想办法去突破它。在坐下来想办法之前，我们往后退一步，仔细考虑这样一个问题：我们到底在优化什么东西？是文档的生成速度？页面资源的加载速度？页面的渲染速度？或者说更高大上的用户体验？这些问题想清楚了，才能分析的更彻底。

其实，大多数的性能优化工作都开始于瀑布流图的分析，下面我们就来看看美团项目详情页的瀑布流图:

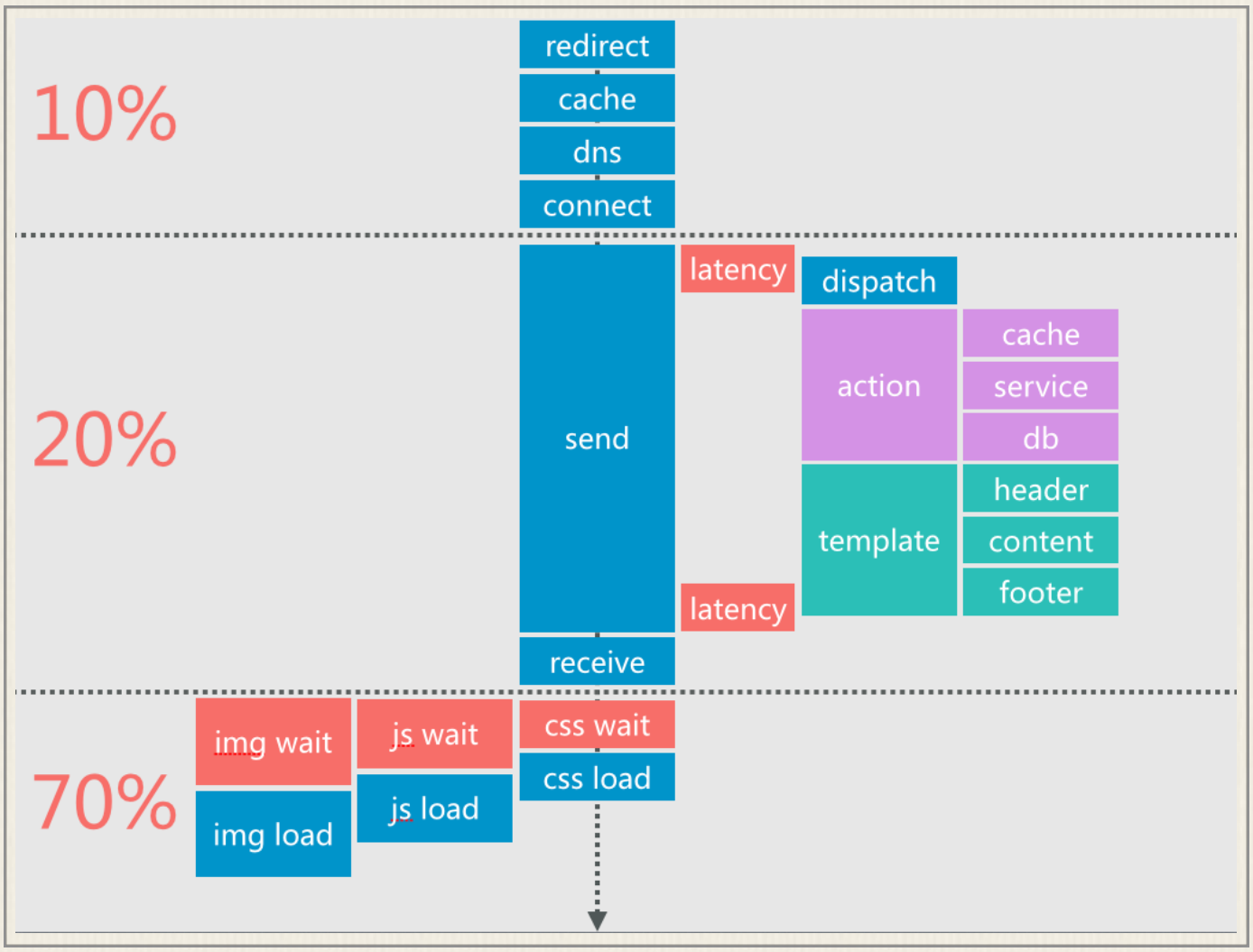


我们把项目详情页的资源分为以下几部分：

- 主文档，即页面的内容，在拿到主文档之前，浏览器啥都干不了；
- 核心 CSS，和首屏图片，在拿到这些之后，浏览器可以开始渲染了；
- 核心 JS，拿到这些内容之后，页面的交互被丰富，但是也会阻塞；
- 其他内容，比如雪碧图，统计脚本等；

从技术上来讲，我们优化的就是这个瀑布流图的每个环节，那么瀑布流图的背后是什么？

其实就是页面加载过程中各个资源的加载时间分解：从上到下的箭头表示时间轴，从浏览器跳转，缓存检查，再到 DNS、TCP 建连，然后发起主文档请求，再到接收完最后一个字节，再到浏览器开始CSS、JS、图片的下载，最后是页面渲染和交互响应。



根据《高性能网站建设指南》上的数据以及我们的观察，整个页面的加载可以划分为 3 大块：网络时间、后端时间、前端时间，发生在网络和后端的时间占到整体加载时间的 10% 和 20%，而前端资源加载时间占到整体加载时间的 70% ~ 80%。

前端资源加载是否快速对性能影响是最大的，这里面资源的加载顺序，并发数量，都有很多的工作可做：比如，如果你发现 CSS 加载之前的阻塞时间很长，那很可能是资源加载顺序不合理，这必然会导致浏览器渲染延后。

页面的加载时间还能分解的更细么？到目前为止，我们都是站在浏览器的视角，划清了各个环节。浏览器拿到文档之前，是不会做任何事情的，后

端响应速度 的变动多数时候能引发性能上的蝴蝶效应，我们的突破口就在后端处理时间上：服务器收到请求之后，会经历请求分发、业务逻辑处理、文档生成这三个阶段，在业 务逻辑处理阶段，会涉及到和数据库、缓存以及内部服务的通信，拿到所有的数据之后，渲染模板，最后发送给浏览器。

对页面加载过程中涉及到的所有环节进行分解和细化，就形成了我们的分析框架。

如何把控性能?

有了分析框架，那么如何全面的把控网站的性能呢?

基于这个框架，我们通过统计脚本加上必要的数据统计（这里的统计都是过程指标，只反映页面加载过程中某个环节的健康状况），就能获得对整个网站的很多内窥。

具体来说，我们对数据的要求是这样的：整个流程各环节的，多维度（比如分页面、分地理区域、分浏览器）的，实时的（方便我们快速实验）。所有的数据都必须是能够反映整体的统计量。

而对于统计脚本，需要满足两个条件:

- 避免对业务代码的入侵;
- 不影响被测量的页面的性能;

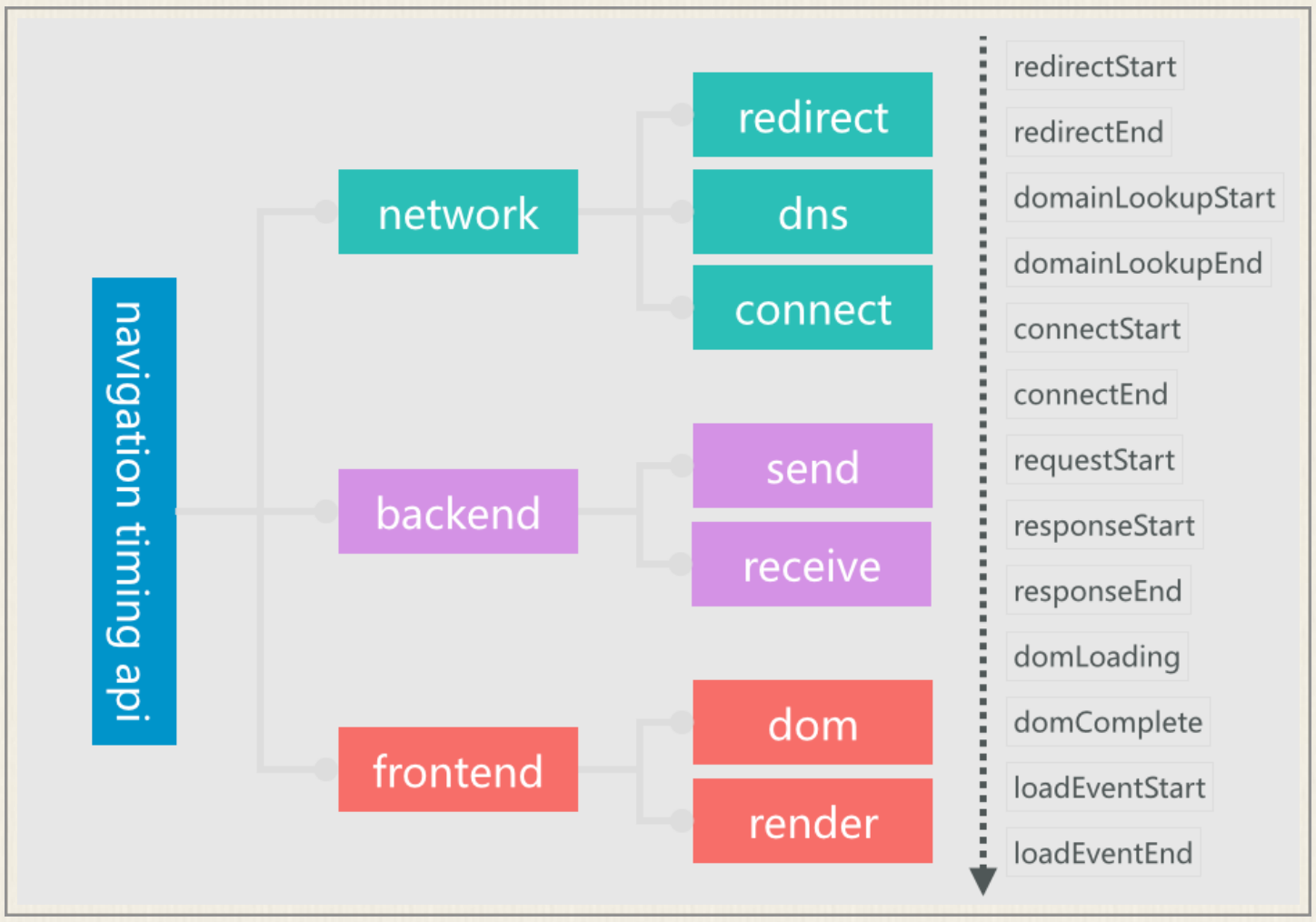
针对第 1 个要求，需要开发独立的统计脚本，避免其与现有的框架耦合，方便移植到其他项目；而针对第 2 个要求，需要在主文档加载完毕之后，再注入统计脚本收集数据，并且尽可能的合并数据请求，减少带宽消耗。

确定了数据统计脚本的约束条件之后，我们从哪里得到这些数据呢？目前使用的主要途径有:

- 主文档加载速度，利用 Navigation Timing API 取得;
- 静态资源加载速度，利用 Resource Timing API 取得;
- 首次渲染速度，IE 下用 msFirstPaint 取得，Chrome 下利用 loadTimes 取得，我们的 Chrome 浏览器用户占比超过 70%;

- 文档生成速度，则是在后端应用内打点来获得；

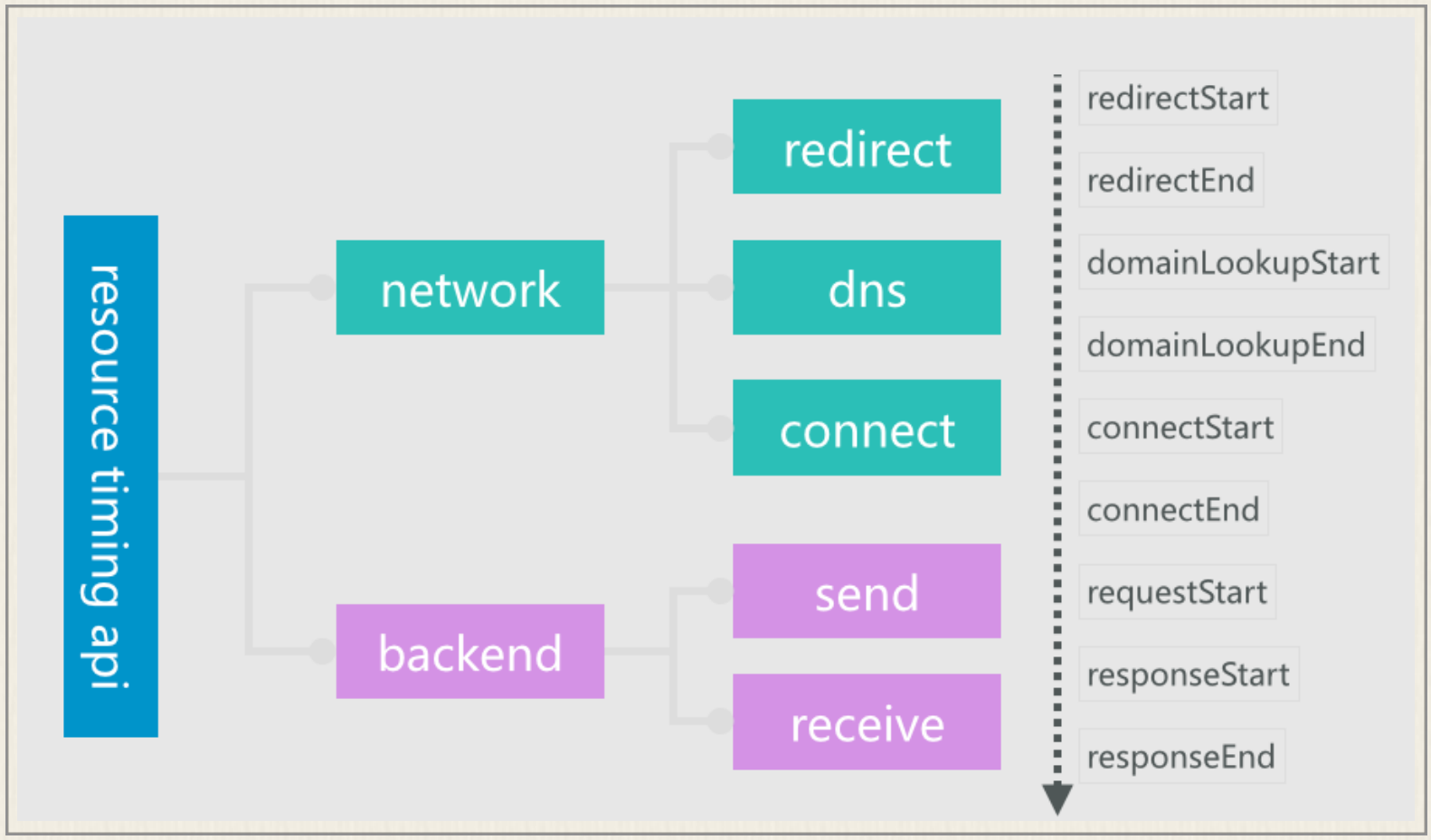
对于主文档加载速度，我们从宏观到微观的做了这样的分解，从上到下的时间流，右边的时刻标记了每个指标从哪里开始计算到哪里截止，比如，跳转时间 redirect 由 redirectEnd - redirectStart 计算得到，其他的类推：



采集主文档加载速度的具体做法是：

- 在主文档 load 之前提供可缓存数据的接口，方便在统计脚本载入前就可以准备数据；
- 在主文档 load 之后注入数据收集脚本，该脚本加载完成之后会处理所有的数据；
- 利用 Navigation Timing API 收集计算得到上图中的指标；
- 给所有数据打上页面、地理位置、浏览器等标签，方便更细维度的分析；

对于静态资源的加载速度，我们也做了类似的分解和采集：



需要特别提示的是，如果你使用 CDN 的话，需要让 CDN 服务商加上 Timing-Allow-Origin 的响应头，才能拿到静态资源的数据。

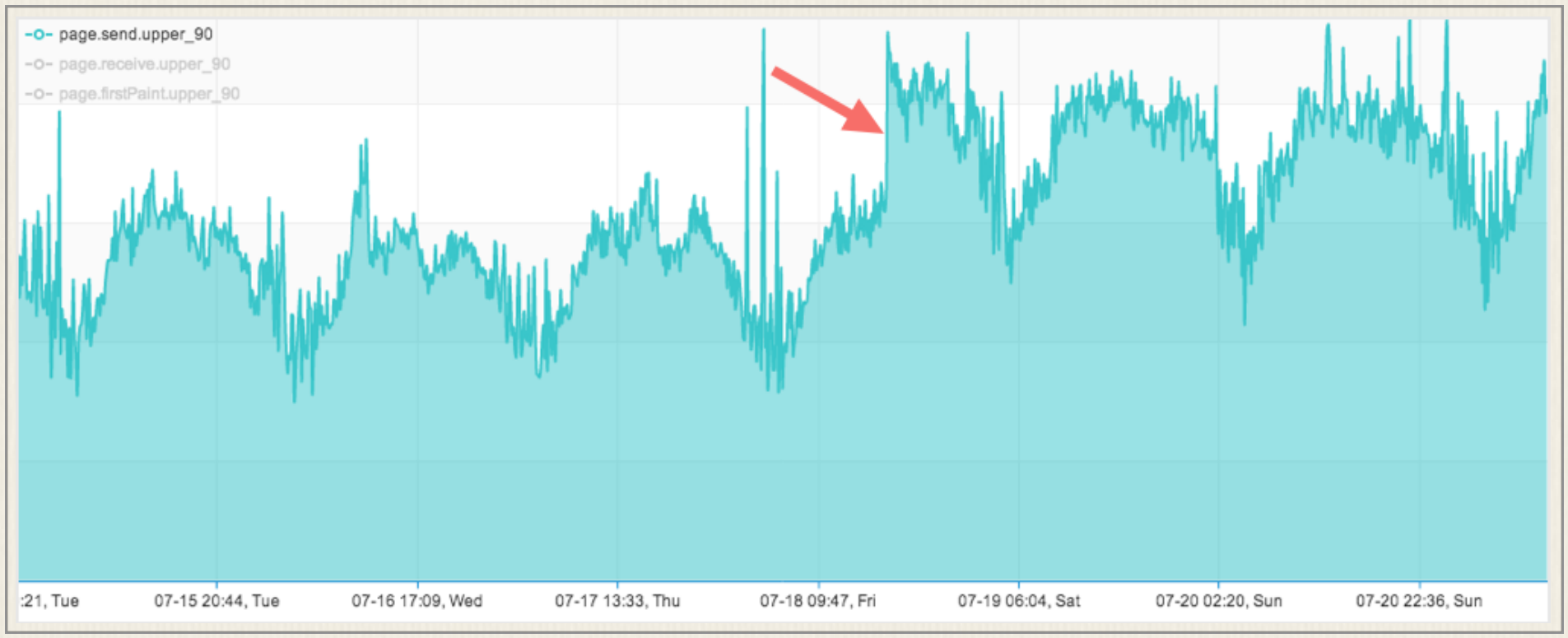
而对于主文档生成速度，我们则开发了性能统计的 Library，在框架级别集成后端性能的时间指标。

实际效果如何?

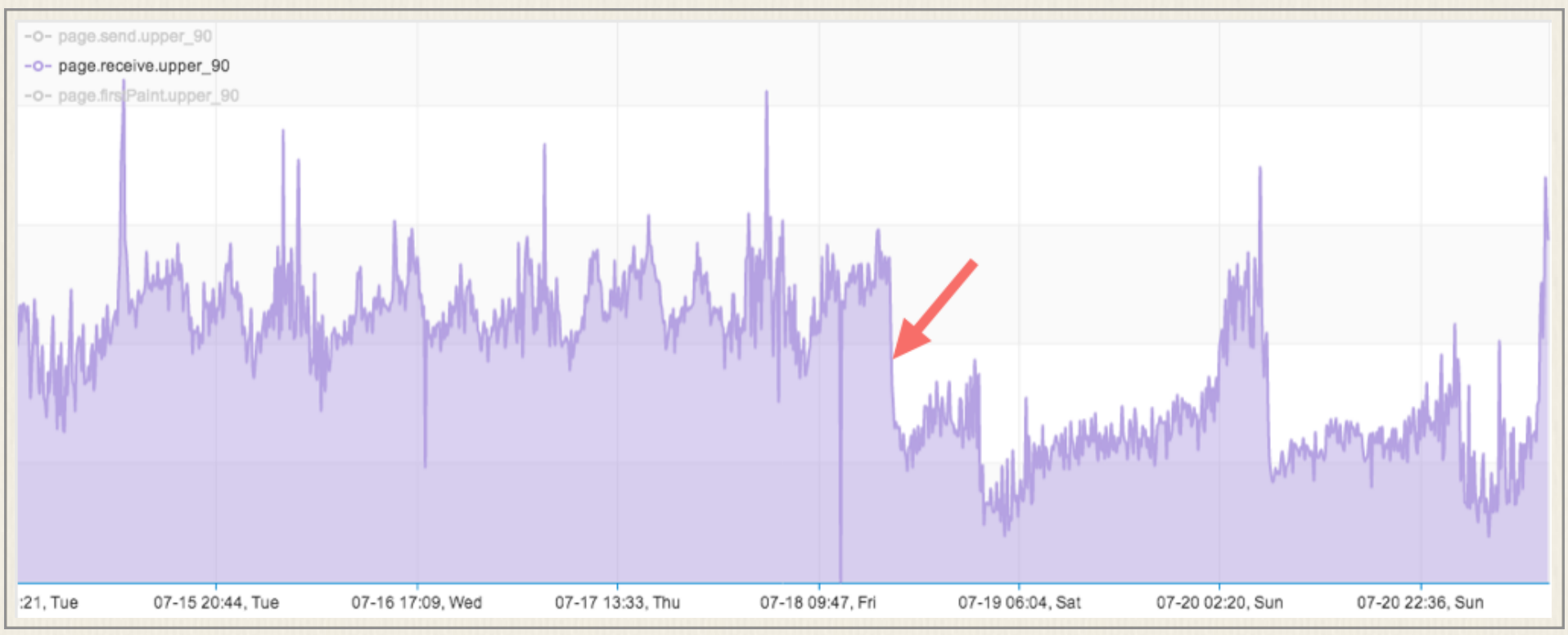
通过上面的各种数据采集，我们拿到了页面加载全流程、全方位、多角度的真实用户数据，有这些数据之后，我们能做什么呢？之前遇到的瓶颈不再是瓶颈，因为我们可以利用这些数据做很多事情，下面举几个实际的例子：

Flush Early 是否有效?

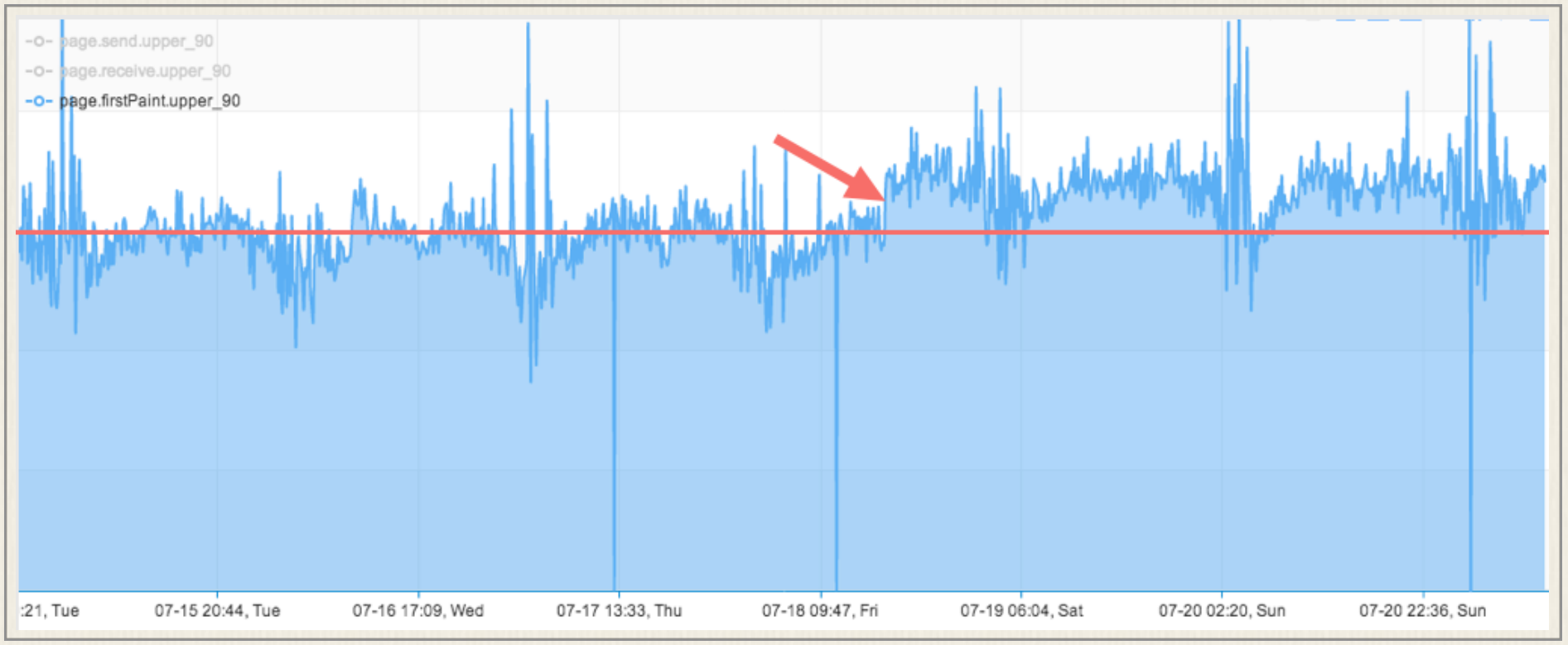
《高性能网站进阶指南》上提到要尽快输出文档的第首字节提高性能，我们很早的时候做了这个事情，但是从数据上看，在页面完全加载时间上的收益不大， 做了更细的数据采集之后，我们快速的在线上做了这样的实验：在特定页面把 Flush Early 关掉，结果发现，浏览器接收到第 1 个字节的时间增加了 100+ms，如下图（红色箭头表示变更上线时间点）：



而完成文档传输的时间减少了 150+ms，如下图：



表面上看，似乎禁用 Flush Early 效果好些，但是再看看浏览器的首次渲染时间，增加了 300+ms，如下图：



也就是说，有些优化措施，总结果指标上看貌似没啥效果，但是换个角度看效果非常明显。有了全方位的数据，我们能更高效的试错。

发现新的优化点

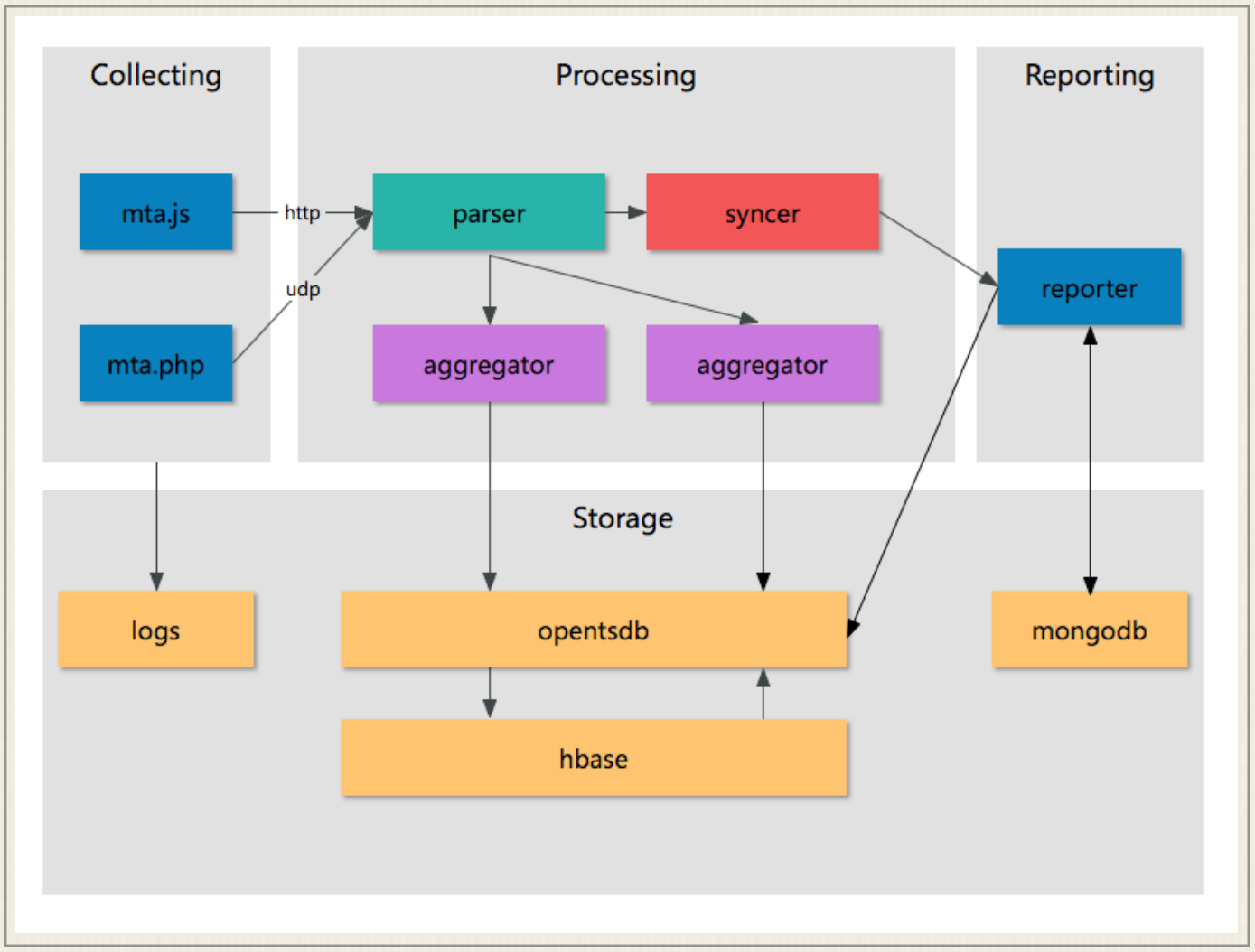
为了优化文档生成速度，我们一度想到优化函数级别的调用，利用 Face-Book 的 HipHop 为 PHP 加速，通过数据发现，在我们生成文档的时间构成中 30 %是在和缓存交互，这个比例太高了，当优化缓存服务器之后，后端时间大幅下降，缓存占比降到 10% 以下。

另外，美团主站的迭代速度非常快，每天大概 50 次左右的上线，通过数据发现，每次上线都会导致性能的轻微恶化，如果某天上线次数越多，那么性能就好不到哪里去？原因我们合并了大量的 JS 请求，当其中的某个模块在某次迭代中被修改，整个合并的文件需要被重新下载，这就对模块拆分和加载提出了更高的要求。

有了更细节的数据我们能有效发现新的优化点。

性能监控平台

我们不光突破了之前遇到的瓶颈，实际上，我们走的更远，因为我们觉得解决一个问题不如解决一类问题，我们解决问题的思路和工具同样能适用于公司的其 他产品线：于是我们在做性能优化的过程中逐步建设起来性能监控平台，目的是为公司的其他产品线和内部系统提供一站式的性能数据收集、计算、存储和展示服务。



目前性能监控平台已经接入 20 余个公司内部系统，能够支持任意指标、任意维度的实时数据查询。该平台为不同的项目提供了性能仪表盘功能，方便快速了解整体的性能状况：

同时还为做性能优化的工程师提供了简单的数据分析功能，方便其以数据驱动的方式的开展性能优化工作：

总结

以上，就是我们做性能优化时遇到的问题，以及解决的办法，下面大概说下，我对这些事情的总结：

- 首先，需要深入的剖析问题，性能分析问题的框架，让很多死角暴露无疑；
- 其次，在性能优化这件事情上，只关注结果指标是不会给你多大帮助的，如果想真的优化，你需要测量过程指标，从过程指标发现更多；
- 再次，解决一个问题比如解决一类问题，解决问题的思路和工具可以沉淀下来，服务更多的团队和同事；

原文链接：http://tech.meituan.com/performance-framework-and-platform.html

腾讯TDW千台Spark千亿节点对相似度计算

作者：腾讯大数据

相似度计算在信息检索、数据挖掘等领域有着广泛的应用，是目前推荐引擎中的重要组成部分。随着互联网用户数目和内容的爆炸性增长，对大规模数据进行 相似度计算的需求变得日益强烈。在传统的MapReduce框架下进行相似度计算会引入大量的网络开销，导致性能低下。我们借助于Spark对内存计算的 支持以及图划分的思想，大大降低了网络数据传输量；并通过在系统层次对Spark的改进优化，使其可以稳定地扩展至上千台规模。本文将介绍腾讯TDW使用千台规模的Spark集群来对千亿量级的节点对进行相似度计算这个案例，通过实验对比，我们优化后的性能是MapReduce的6倍以上，是GraphX的2倍以上。

一、介绍

相 似度是指两个节点之间特定属性的相似程度，相似度计算是数据挖掘、推荐引擎中的最基本问题。例如在推荐系统中通过计算推荐物品的相似度，从而给目标用户推 荐与他喜欢的物品相似度较高的物品，或是计算用户之间的相似度，给目标用户推荐与其相似的用户喜欢的物品。因此，相似度计算技术在很大程度上决定着推荐系 统的性能。

随着大数据时代的来临，日益增加的数据量使得单机的计算能力已经远远无法满足需求。在对大规模的节点对进行相似度计算时，分布式 处理往往是可行的解决方案。MapReduce是目前流行的分布式编程框架。Hadoop与Spark是MapReduce编程模型的两个开源实现。相比 于Hadoop，Spark提供了cache机制，增加了对迭代计算的支持；还提供了DAG调度来支持复杂的计算任务，减少了中间结果的磁盘读写，能够获 得更佳的性能。

本文将介绍腾讯TDW使用Spark来对千亿量级的节点对进行相似度计算的案例研究，我们在计算方法和系统两个层次都进行了 改进优化，获得性

能提升的同时，还具备了千台集群的扩展能力。在本文接下来的内容中，先简要介绍相似度计算这个问题，接着介绍在Hadoop和Spark 上的两种实现方式，并通过在两个数据集上的实验进行对比分析，最后进行总结。

二、问题描述

输入数据可以表示成两张表：

1.节点关系表relation，字段有id, fid，表示两个节点存在关系。

2.节点特征表features，字段有id, feature，表示每个节点具有的特征信息。

下列两个表格表示了在一个拥有6个节点的关系网络中，节点关系表和节点特征表的情况。

id	*fid*
2	1
4	1
1	2
5	3
6	3
6	5
5	6
3	6

Table 1： relation

id	feature
1	$(x_1, x_2, \dots, x_{n_1})$
2	$(x_1, x_2, \dots, x_{n_2})$
3	$(x_1, x_2, \dots, x_{n_3})$
4	$(x_1, x_2, \dots, x_{n_4})$
5	$(x_1, x_2, \dots, x_{n_6})$
6	$(x_1, x_2, \dots, x_{n_7})$

Table 2： features

相似度计算即是对节点关系表中的所有节点对 (id，fid)，其特征向量分别为 和,利用相似度计算函数similarity-Calculation，计算和之间的相似度。相似度计算函数similarity-Calculation依据具体的相似度衡量方法而定。

三、MapReduce 解决方案

Hive是建立在Hadoop之上提供SQL接口处理的海量数据处理工具，对于上述相似度计算问题，其计算流程可以用如下SQL来描述，并使用Hive来计算。

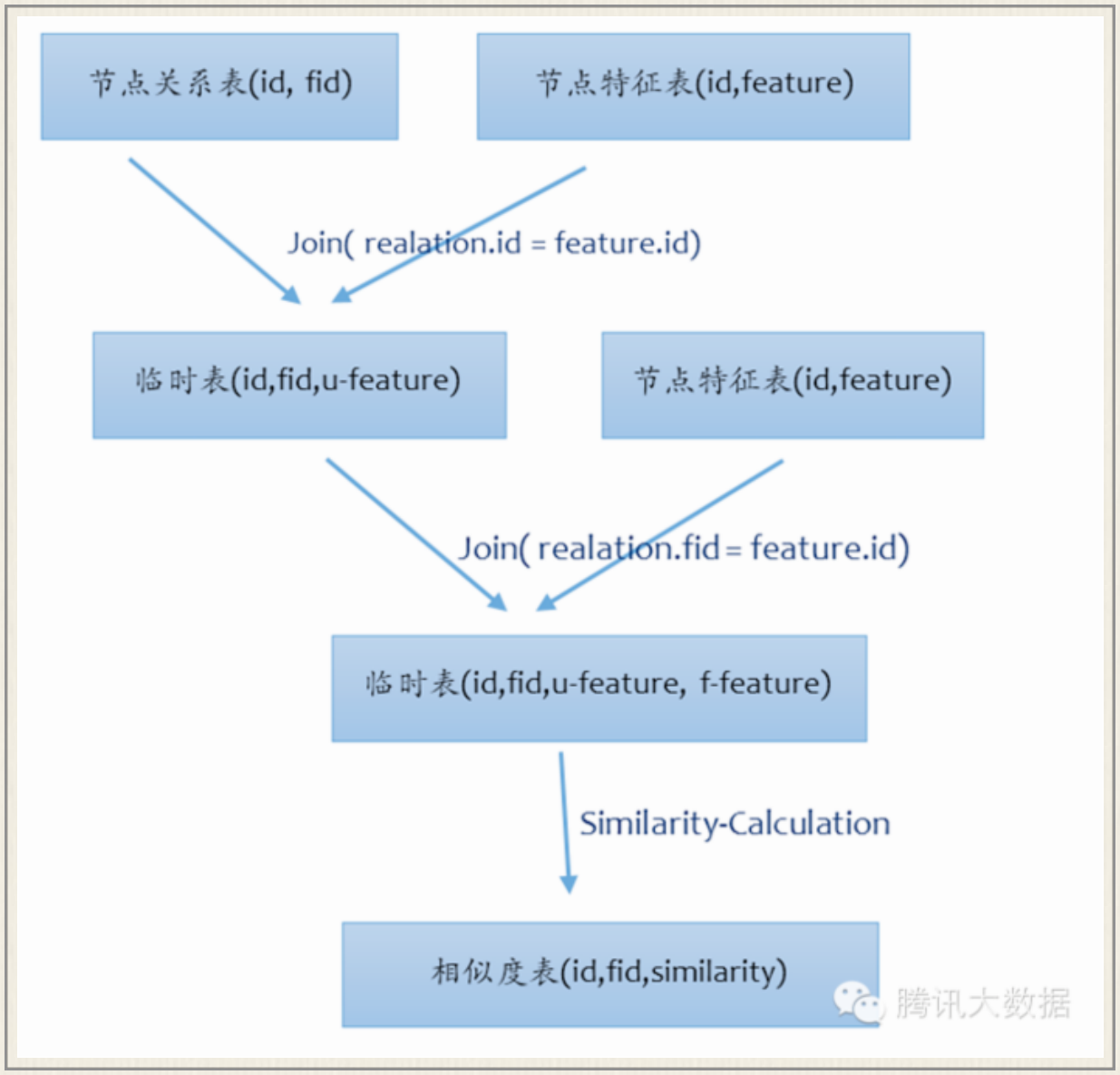
```
SELECT t1.a,t1.b, similarity-Calculation(t2.feature, t3.feature)

FROM relation t1 JOIN features t2 ON (t1.id = t2.id)

        JOIN features t3 ON (t1.fid = t3.id)
```


整个计算流程可以分为两个步骤：

1.通过两次JOIN操作，生成一张临时表，临时表中的一个元组对应节点关系表中的一对节点和这两个节点的特征向量。

2. 遍历临时表，对每个元组中的两个节点计算其相似度。

下图展示了该SQL语句的执行过程：



使用Hive对千亿节点关系记录进行相似度计算，两次JOIN操作成为性能的主要瓶颈瓶颈。在两次JOIN的过程中，网络数据传输和磁盘读写达到了200TB，集群多数结点的硬盘无法支持，任务失败经常发生，作业运行了时间超过了24小时。通过将节点关系表拆分成多个子表，每个子表独立地进行相似度 计算，多个子表的任务并行执行，最后再将多个子作业的结果汇总，得到最终结果。采用这样的方式，作业总时间仍然超过了24小时。

四、Spark解决方案

通过对Hive计算过程的分析，我们发现网络数据开销主要来自于节点特征向量的大量复制。对于节点关系表中的每对关系，计算时都需要得到两个

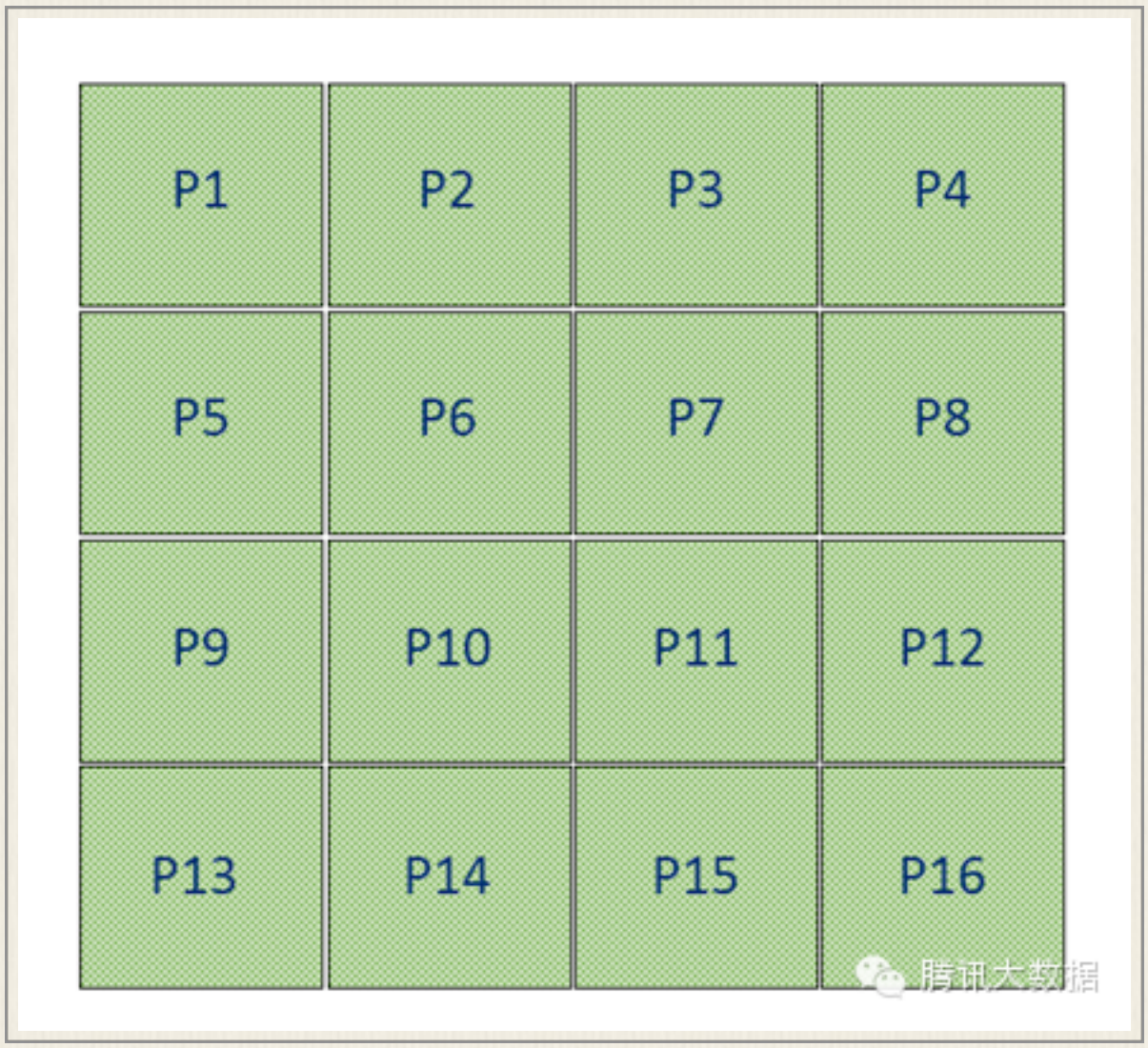
节点的特征向量，从而导致了大量的数据复制。因此，我们从两个方面去减少数据复制：

1.采用二维图划分的思想，减少节点的复制数目

2.每个数据分区中，对于同一个节点，只保留一份该节点特征向量

二维图划分方法

任何一张关系网络，都可以用一个大矩阵M来表示，矩阵的两个维度用来表示节点，矩阵的元素M[i, j]表示节点i和节点j是否存在关联，如果存在，则M[i,j]值为1，否则，M[i, j]值为0。下图展示了通过采用二维划分的方法，将一个矩阵划分成了16个分区。



使 用二维划分可以减少节点的复制数目。假设分区总数为，采用一维划分的方法，最差情况下每个节点的复制份数是，即每个分区都会有该节点的复制；采用二维划分 方法，最差情况下每个节点的复制份是 。对于大数据量，分区总数通常很大，所以采用二维划分通常可以减少每个节点的复制份数。

计算步骤

1.利用二维划分方法将节点关系表划分成多个数据分区，假设我们将分区数设为4，则Table 1所示的节点关系表将会划分到4个分区，每个元组对应的分区如下Table 3所示：

Pid	id	fid
1	2	1
	1	2
2	3	6
3	4	1
	5	3
	6	3
4	6	5
	5	6

Table 3

id	pid-List
1	1,3
2	1
3	2,3
4	3
5	3,4
6	2,4

Table 4

2.根据每个分区中的节点列表，计算出每个节点所在的分区列表，称为路由表，记录了每个节点所在的分区信息，其结果如Table 4所示。

3.根据路由表将每个节点的特征向量发送至每个分区之中，保证每个分区中一个节点只保存一份特征向量，如Table 5所示。

pid	id-List	feature-List
1	{1,2}	1-> X_1 2-> X_2
2	{3,6}	3-> X_3 6-> X_6
3	{1,3,4,5}	1-> X_1 3-> X_3 4-> X_4 5-> X_5
4	{5,6}	5-> X_5 6-> X_6

Table 5

4.对于每个分区，将该分区的关系集合与该分区中所有结点的特征向量进行关联，遍历每对节点关系，利用相似度函数和特征向量计算二者的相似度。

通过以上步骤，即可以计算出节点关系表中每对节点的相似度。与MapReduce的计算方法相比，如果一个用户多次出现在同一个分区中，比如用户1在分区1 中出现了两次，上述计算步骤只会将用户1的特征向量发送一份到分区1中，但是MapReduce的计算方法会发送两次，产生冗余的网络数据传输。使用上述 计算方法，我们将网络传输量降到了50 T，远小于MapReduce方法的网络传输量。

系统层次优化

除了在计算流程上进行改进，我们还对Spark进行了以下方面的优化：

1）优化分区参数设置。在相似度计算的应用中，分区个数越多，会导致节点的复制份数增加，从而增大网络数据传输量。因此我们基于中间结果的统计信息来确定确定分区个数，使得在充分利用每个节点内存和CPU的前提下，最小化分区个数。

2）优化内存表示。由于数据量大，对象个数多，导致内存使用量较高，GC时间较长。我们使用列存储格式来对内存数据进行压缩，减少数据量的同时也减少了对象个数。

3） 提高网络稳定性。随着集群中机器数目的增加，网络连接数也会成倍增加。当网络出现拥挤时，经常会伴随着连接超时从而导致shuffle数据拉取失败。更糟 糕的情况是，网络超时会让Master误认为Executor已经丢失，故会使得整个Executor上已经完成的任务全部重做。因此在shuffle时增加网络超时重试机制，同时控制每次发送的请求连接数，避免shuffle拉数据超时，减少任务失败次数，防止Executor丢失的情况出现。

4）使用sort-based shuffle时将文件块索引信息缓存一份在内存中，后续拉数据时直接读内存获取索引信息。预测执行时，当同一任务的一批运行实例有一个完成时，杀掉正在运行的其余实例，提早释放计算资源。

5） 参数调整。由于每个Executor进程还会使用到堆外内存，因此Executor进程占用的内存往往会大于JVM设定的最大值，为了保证Gaia不

会将 超过JVM内存的Executor进程杀掉，配置参数spark.yarn.executor.memoryOverhead以免被kill。由于 Executor在Full GC时需要较长时间，需要配置参数spark.storage.blockManagerSlaveTimeoutMs来延长blockManager的 超时时间。

五、实验对比

实验环境：

我们分别在拥有200台、600台和1000台TS5机器节点的集群上进行了对比，每台机器拥有64GB内存，2*12T硬盘，24线程CPU。

我们在两个数据集上进行了Hadoop、社区GraphX和TDW-Spark的性能对比，一个数据集拥有五百亿节点对，而另一个拥有千亿量级的节点对。实验结果如下表所示:

关系表记录数/机器规模	Hadoop	GraphX	TDW-Spark
五百亿(200 节点)	> 12 hours	5 hours	2 hours
五百亿(600 节点)	-	-	55 mins
五百亿(1000 节点)	-	-	43 mins
千亿(200 节点)	> 24 hours	15 hours	7 hours
千亿(600 节点)	-	-	2.5 hours
千亿(1000 节点)	-	-	2 hours

通过上述实验对比，可以看出在MapReduce上的实现的性能远远低于在Spark上的性能，使用JOIN的方法使得网络通信开销非常大，五百亿 数据集的任务执行时间超过12个小时，千亿数据集任务执行时间超过24个小时；GraphX采用的同样是二维图划分，但是由于其是一个面向通用的图计

算框 架，维护了复杂的数据结构和计算流程，造成性能下降。同时，GraphX在网络稳定性方面存在许多问题，当集群规模达到600台时便会有大量的任务失败。

与前两者相比，TDW-Spark在集群为200台时在两个数据集上都获得了较大的性能增长，所消耗时间少于GraphX的一半。当集群规模从 200台扩充至600台，TDW-Spark在五百亿节点对数据集上获得加速比218%，在千亿节点上的加速比为280%；当集群规模从200台扩充至 1000台时，加速比分别为279%和350%。因此，TDW-Spark不仅在性能上获得了很大的提升，还可以在千台规模的集群之上稳定运行，同时获得 良好的水平扩展能力。

六、结论

Spark是目前Apache中最活跃的开源项目之一，已经形成了一套成熟的大数据处理生态系统，为大数据处理提供了强有力的支持。TDW目前维 护了上千台的Spark集群，支持了公司多个业务的挖掘分析和实时计算类任务，我们会在易用性和稳定性方面进行进一步的改进和优化，构建强大的大数据处理 平台，给业务提供更有力的支持。

原文链接：http://mp.weixin.qq.com/s?__biz=MzA3MDQ4MzQzMg==&mid=216841618&idx=1&sn=c3ccf6352a2b6d9b6da194ce4b6a1db8&3rd=MzA3MDU4NTYzMw==&scene=6#rd

程显峰：IT病得有多重？技术圈交际花谈研发管理怪现状

作者：技术人攻略

导语：本期采访对象程显峰@程显峰-Mars，蓝海讯通COO。素有“技术圈交际花兼娱记”称号的显峰，是MongoDB中文社区发起人，曾任积木盒子技术副总，Admaster首席布道师，混迹于安全、广告、云计算、大数据、互联网金融等多个技术圈。

“什么时候采访我？”在GitCafe北京分部的开幕活动上，显峰半开玩笑半认真地问我。认识显峰的时间不算短，总在各种技术大会与小会上频繁碰面。过去一年多，他的工作状态算不上“稳定”，这不，刚离开高大上的互联网金融，投身APM的伟大事业。真要采访，得先选出一个可行的话题切入角度，他引 用《人件》里提到的“高科技幻觉”，从传统工程的角度谈了谈对IT研发管理的看法，于是采访主题顺利地定了下来。

这通吐槽显然憋了很久，不乏“这个行业充满了骗子与强盗”的激烈言辞。我有点被惊到，常看他以典型的东北式幽默与人调侃，并不知道他原来如此严肃。在北五环外的东升科技园，我们从下午两点半，一直聊到天黑，期间换了3个场地。末了，他叹到：“为什么好好做技术这么难！”

显峰无需借此博眼球、搏出位，公开发表这些话给他带来的潜在风险远大过收益，观点犀利自然会赢得一些赞同，也难免招致对号入座的无端恨意。在乌合之 众汇聚成的网络空间，谩骂而非理性的讨论是更为常见的交流方式。虽然在文字上做了些处理，我仍然对其可能带来的争议无比担心。发给他确认，几乎没做大的修 改，仅回了个：“整体很流畅，但细节上的文字还不够平滑。”看，真是个对品质要求很高的人。

这些细节暴露了他的始终如一，也让我更加理解显峰的选择，不安定的背后，自有他严格的价值坚守。如果有机会，显峰希望去教小孩写程序，热爱学习的人是真诚的，他喜欢和这样的人在一起。

技术人攻略：你从什么时候起开始对对研发管理感兴趣?

我是学工程出身，本科就读于哈工大航天工程与力学系，研究生是悉尼大学的航天专业，期间受到了严格的工程训练。传统航空业的研发和制造体系非常 完整，拿造飞机举例，悉尼大学的本科生就完全可以组装出可供销售的飞机，因为整个生产过程非常严格，任何一个扳手都有编号，有详细的记录和流程，不可能搞 错任何东西。

虽然专业选择了航天，我对编程却非常热爱。从小学就开始写程序，那时候家里没有电脑，每次上机需要走40分钟山路。研究生期间，独立完成了完整的有限元分析软件，算是我在科学计算领域的一次实践。

回国之后，我加入的第一家公司Antiy，很重视底层技术，产品做得非常成功，但研发管理做得并不好。我在那期间学了很多软件研发历史，但在研发体系建设上，还是留下了许多遗憾。随后加入做互联网广告监测的创业公司AdMaster，当时公司正在筹建，人员来源多种多样，研发管理问题比较突出。 我的职位是专职敏捷教练，配合技术负责人做团队建设，开始更深入地思考研发管理。

刚进入IT这一行时我很难理解，为何在传统工程和制造领域很平常的事情，在IT领域却是需要商量和悬而未决的。可靠性在航天等领域早已解决得很 好，为何软件行业却一直解决不了产品质量问题。后来看了不少管理的书，发现IT研发管理的许多思想都是从建筑业、制造业借鉴而来，例如快速迭代、精益管理 等概念。

结合工作实践，我逐渐发现了研发管理问题的症结所在。研发能力是工作的综合体现，内功水平是关键，任督二脉打通了，练什么都很快，至于到底用哪 个套路，是很轻松的一件事。举个例子，大家通常说要做“敏捷转型”，认为自己是从传统软件研发转型成敏捷，关于二者的争论也显得像是泾渭分明的两派，但实 际上不是。难道传统软件就不做配置管理吗？难道敏捷就不做测试吗？这两派理论有八成是一样的，即便在软件工程教科书里，也同样有关于质量控制、配置管理、 迭代等理论，如果很好地去执行，同样可以达到不错的效果。

为什么敏捷转型失败的案例很多，因为企业并不具备相应的内功，只想寻求解药，以为敏捷能有所帮助。实际上如果不打好基础，结果还是一样。具备这种内功的人，玩传统软件也会很好。航天、制造、金融行业并不过分

强调敏捷，当然敏捷里的好东西，他们也能非常快地去借鉴。《精益软件开发艺术》这本书的作者来自波音公司，他们将其在制造上的经验应用于研发，对软件的驾驭能力相当高。

强调时髦的概念，对研发帮助并不大。比如知道了TDD测试驱动开发，对团队帮助有多大呢？TDD想执行好，要求对测试理论有深入理解，但大部分国内开发团队不仅不具备很高的测试水平，连测试是什么，如何测都不知道。这种情况下去推广TDD是没有意义的。

技术人攻略：根据你的观察，国内研发管理有哪些常见问题?

我观察到国内研发管理主要的问题有几个：第一是过于强调个性，缺乏共同价值观；第二是内功差，不重视软件质量；第三是很多从业人员眼界狭窄，拿无知当个性；第四是对技术缺乏敬畏之心；第五是整体气氛浮躁，擅长炒作概念而非脚踏实地做事。

IT这一行太推崇个性，过于强调创新，强调极客，而对于共同价值的坚守非常少。传统工程领域里，大家都遵照明确的规范和标准做事。软件行业的国 家标准很落后，大家也都不执行，几乎每个公司都会自定义一套方法和流程，大家各说各话。个性的东西太多，达成共识的东西太少，导致软件行业的人很难树立共 同价值观，以及清晰的研发过程。

我做软件咨询的时候发现，不少合作多年的团队，都未能在基本价值观上达成一致。例如自家产品到底能解决客户哪些问题，10个人能给出8个答案。 我认为研发管理首先要解决的问题，是形成一个团队，这就要求大家必须有足够多共性。想要塑造有战斗力的团队，需要模仿军队管理，大家穿一样的衣服，迈一样 的步子，用同样的方式使用工具，减少不必要的浪费和沟通。

建立共性的关键之一，是要对代码质量树立共同的认可规范。好代码必须干净、可维护、可测试性好、适宜阅读。如果在大规模项目之前没有就此达成统一，大家冲上去的时候，再说如何配合、包抄，只会被打得一败涂地。

关键之二，是要做好版本控制。版本控制是研发的基石，开发人员每天都要用，而即便很多资深程序员，对版本控制的使用方式依然很落后。版本

控制最 基本的要求是可回滚，但国内大部分公司做不到这一点。《精益软件开发艺术》这本书第0条就讲：代码必须在版本控制工具里。离开这个基础，其它的改善都是无 用功。我原来一直在推Git，本质原因还是我们的内功特别落后，你看Github有多流行，就知道国外做得有多好。

技术人攻略：国内研发管理内功不足，除了版本控制，还体现在哪些方面?

除了版本控制外，调程序和测试的情况也不乐观。国内程序员调试程序大部分全凭拍脑袋，不能以程序的方式思考问题，不仅不具备调高难度算法的能力，也没有清晰思路去解决问题，更不会使用工具。

在互联网领域，测试的重要性远远被低估。合格的测试开发工程师应该既懂测试，又懂开发，还要能教育其它开发工程师。这种人在现实情况下很难找到，根据我面试的经验，能把最基本的单元测试要点说清楚的人都不多。

做互联网金融这段时间，我接触过国内很多第三方支付，都在测试上做得一塌糊涂。举个例子，开放平台让商家接入之前，需要提供一个虚拟测试环境。 Paypal的正规做法，是给每个商家建立一个沙盒。而国内大部分厂商的做法，是让所有商家共用一个测试账号，往里面打一分钱。这一看就根本不懂测试理 论，沙盒测试是标志性的东西，如果你到某个医院，发现那里没有显微镜，那就一定说明这个医院不具备做某些类型化验的能力。

电信、金融、制造业等传统软件开发领域，对软件质量重视程度很高。互联网领域最不重视软件质量，普遍采用的灰度测试，虽然能解决体验、交互流程 上的问题，但并不能解决质量和正确性问题。测试能力是很基本的内功，做灰度可以，但不能对测试一窍不通还无所谓。这好比你有10发子弹，因为时间、资源所 限，只能打1发。但如果你只有1发子弹，你就打，不要说别的不好，因为你根本不知道完整的方式该是怎样，只能灰度。

国内的创业者天天看TechCrunch，知道美国的市场、机会、商业模式，唯独别人的研发流程不了解，所以只会抄袭一些表面的东西。媒体总是 报道Facebook一夜成名，但很少有人知道，在这家公司刚开始壮大时，就从Mozilla挖了一位非常资深的专家去负责工程。这些经验丰富的人是团队 的

定心丸，前进路上有多少坑，他们早就踩过了。研发有本质的客观规律，不能因为你年轻，你创新，就逾越这些规律。

技术人攻略：你提到的从业人员眼界狭窄，表现在什么地方?

从业人员不怎么看书，是这个行业普遍存在的问题，最多看点讲程序开发的书，所以有文化的程序员特别少。作为完整的人来讲，基础文化结构的缺失，导致大部分程序员看问题很偏激，没有常识，不知道历史，还总拿无知当个性。

例如做技术选型时，看好某门技术，就要用到项目里，这其实是非常幼稚的行为。技术选型一定要考虑团队的驾驭能力，考虑能不能持续招到懂这门技术的人，以及最重要合作伙伴用了哪些技术，你选的这门技术能不能同他们高效沟通等非技术因素。

研发管理90%的问题，30年前在美国已经出现过，好好看看经典书，90%的问题能解决得挺好。不要总觉得自己是世界上第一个遇到这个问题的人，差不多你都快成为世界上最后一个遇到这样愚蠢问题的人了。干了多年研发管理的人，都没读过研发管理经典的书，是很可笑的事。

我经常说，想去研究软件考古学。软件的历史比较年轻，可考证的东西又比较多，能研究出相对清晰的历史、来源、派系，帮助我们了解行业的发展过程。《人月神话》这本软件工程经典书，就是讲软件开发的历史，程序员知道历史后，会更有兴趣去思考整体的行业脉络。

上大学时，我们会从各种空难事故中，学习飞机设计失败的教训，例如某个部位为什么要这么设计，是从哪一次空难开始改进的。航空工业能发展成现在 这样，不是由几个小屁孩拍脑袋做成。在大体理论框架没有突破的前提下，许多改进都是基于已有经验，对细节的精益求精。任何行业都需要积累，研发管理也类 似，我们需要对行业历史有所了解，要传承，而不是完全去创新。计算机行业理论框架突破并不多，并且能得图灵奖的理论，跟大部分撅着屁股干活的人没啥关系， 所以还是老老实实把这些经典理论继承了，对你的帮助更大。

至于为什么要读那么多其它方面的书，除了能提高人文素养，还能帮你解决自身的问题。国外软件业大师，在思考自己行业问题时，常常能旁征博

引其它 行业的案例，例如引用一本护士学的书、一本机车修理的书，或一本建筑电气的书。各学科反复交叉会带来启发性思考，可能你这个行业的难题，在其它行业就不是 事儿，帮助你开拓新思路。

技术人攻略：对技术缺乏敬畏又如何理解?

国内一些程序员懒、没有开阔的视野，对于技术缺乏敬畏之心，觉得自己什么都懂，不需要特别谦虚去学技术，一幅老子写代码天下最牛逼的样子。问他行业里有没有偶像，回答没有，问他知道业界谁做过什么东西，回答不了解。这种人是行业祸害，拉低了行业平均水准。

开发人员能不能成长，只要看有没有追求就可以。面试时我通常会问几个问题，例如最近学了什么？通过什么途径去学习？看哪几本书？都是谁写的？他 还写过什么书？关注什么开源项目？谁写的？他还做过什么项目？这几个问题如果能很清晰回答出来，说明面试的这个人是有追求的，起码有吹牛的追求。如果一个 程序员连吹牛的追求都没有，是很失败的。

那这群人为什么如此骄奢淫逸？因为拿钱太轻松了。互联网公司程序员离资本非常近，光今年上市的公司就大概有20家，行业发展得实在太好。国内互 联网已经快15年没有寒冬了，包括2008年金融危机时，企业融资可能受点影响，但程序员的薪水一路水涨船高。除了干IT，有几个刚毕业的人能拿到上万工 资呢？金融行业能拿到，但不需要IT这么多人。

美国每7、8年，就会经历一次经济周期，而国内这一代程序员没有经历过寒冬，所以不珍惜自己的工作，不知道自己真正的价值在哪里。出来混终究是 要还的，大量发行货币必然导致通货膨胀，只是时间早晚问题。从经济学角度讲，市场有了泡沫，需要经历一次大萧条，把泡沫挤压干净，才能变成更健康的环境。 经纬合伙人已经写信，让大家做好过冬准备，如果资本持续注血，繁荣的假象就会持续得更长，如果市场找钱很困难，那大批互联网公司就会死掉，释放出大量人 员，工资水平马上就会下来。

研发工作非常辛苦，需要踏实态度和长期努力，通过日复一日的艰辛劳动才会有所收获。国内浮躁氛围很难培养出好的工程师。但换个角度，工程师价格高了，对真正有乐趣从事这一行的人来说，还是好事。

从趋势上看，技术学习也正在发生变化，在校学生如果心态够开放，通过参与开源社区快速获取经验，在学校里就能练就很好的内功。这群人一旦成为一 个小气候，可以直接和毕业5年左右的人竞争。尤其是当经济形势不好的时候，那些得过且过的人会很危险，终究有一天，他们会拿起书本去学习，知道自己根本不 值那个价。

技术人攻略：这种行业普遍存在的浮躁心态，带来了哪些不利影响?

我们生活在一个非常功利、信仰缺失的时代，人们只想快速获取财富，很难有正确的价值坚持。用博弈论解释，这种浮躁走向了囚徒困境，类似日本、德 国这种成熟社会，大家做事都不浮躁，整个社会能达到比较高的均衡。而在一个浮躁社会，没按规矩的人走得更快，于是那些按规矩做事的人就吃亏了。这种浮躁其 实把大家都害了，把行业也害了。

IT现在和钱离得比较近，所以病得挺重，整个行业里充满了骗子与强盗。大家努力的方向不是提升自己，而是只要能获得钱的事情都会去做。任何时代都有骗子，但一个国度里大部分人都是骗子，是不正常的，还是应该实实在在创造一点价值。

热衷于炒作概念，是行业浮躁的表现之一。前几天参加一个研讨会，讨论了半天，才发现这群人不是在玩大数据，而是玩“数据”。因为以前根本没有数 据，决策主要靠拍脑袋，现在有数据了，就觉得自己与时代划上了等号，想裹着这个外衣去挣钱，真是无知者无畏啊。好多人觉得有Hadoop集群了，上了硬件 了，从政府那里拿到钱就牛逼了。可对数据没有理解，不知道怎么用Hadoop发挥价值，有钱也没有用。云计算也类似，被地方政府当成了房地产来搞，涌进许 多根本不懂这个行业的人。

这种浮躁会导致软件研发竞争优势下降，我们在圈子里有讨论，如果做高端基础软件，硅谷研发成本会比国内更低，能雇佣更高素质的人才，有更好的配 合，以及更确定的产出。国内拿到钱的互联网公司，将来可能都会去北美建立研发中心。贵不贵是一说，还有值不值的问题，为什么中国建研发中心不值，这是一个 很耐人寻味的问题。

互联网行业看上去门槛低一些，创业相对容易，但总要设置一些门槛给竞争对手，所以还是要有自己的积累。我以前曾喷过阿里这样的公司，觉得

他们做 的东西不够专业，但后来我改变了观点，他们能坚持这么长时间，能把云计算推到这样的高度，就算走了一些弯路，也是值得敬佩的。这些实实在在的创业者，才是 业界的良心。

技术人攻略：根据你做敏捷咨询的经验，实施技术团队过程改进的最大困难在什么地方?

最大的困难在于建立起团队成员对你的信任。许多敏捷实施失败的原因，就是因为程序员不信你，特别是团队资深的人不信你，基本一定会失败。从事敏 捷咨询行业的人，许多并不是技术背景，他可以讲一大堆方法论，但程序却写得乱七八糟。所以想做好技术团队的过程改进，至少你得是个懂技术的人，首先要向团 队展示出自己的技术能力，才有机会去解决困扰他们的问题。

管理大师德鲁克曾说过：“你度量什么，就能改善什么”。在具体过程改进实施中，我喜欢从细节着眼，寻找一些善于实施，又能见到效益的改善。比如我经常会用到一个方法：度量程序员的时间花在了什么地方。如果大家都是靠猜测，管理不好也不奇怪。

我曾经装过一个软件，记录自己使用电脑的行为，例如统计每天用了多少时间上微博、聊QQ、写邮件，还是写代码。真正把时间记录下来之后，才发现实际结果和自己的感觉大相径庭。社交要消耗大量时间，非常影响工作效率，所以后来我把IM都关掉，集中处理工作。

一个管理良好的团队也是一样，想要改善需，就必须要有动力。作为管理者，你必须刺激团队成员身上的动力，给他们一面镜子，照出来他们背后的魔 鬼，这样才会有改善的基石，这也是建立信任的其中一步。大部分程序员都很尊重事实，当他发现每天5个小时都花在聊天上，自己就会想办法去改善。逐渐实行这 样能见到效益的改善，就会获得团队信任。

好多找我做咨询的人，容易关注一些表面现象，比如敏捷实施的各种方法。在我看来，表面的东西只占20%，真正想做好过程改进，必须花很多时间做 基础工作。比如上面提到的对团队工作时间的测量，对你的改善目标，提供了强有力的数据支撑，是非常基础的改进工作。敏捷是一个方法

论，在团队内功真正得到 提升之前，那些方法没有任何用处，而且高深莫测的方法，会让大家感觉到不确定，容易对此起反抗。

实际工作中，许多咨询顾问不会花精力，去做看上去没有科技含量、很琐碎、不为人知的事情。就像刚刚谈到的大数据行业现状，大家在会上讨论着大数 据的建模、分析、如何出漂亮报表，但80%的脏活累活，是要把数据有效地进行搜集和整理，它是简单的体力活，但做不好的话，根本没有后面这些故事。过程改 进也类似，绝大部分工作就是普通工作，没有太多技术含量，也没什么值得可说的，但必须把这些基础打好，才会有后面的故事。

国外谈论敏捷的，都是有20多年工作经验的人，敏捷宣言发起者，都是业界大牛。而平时工作中，我常接触到一些许多没什么工程经验的人在实施敏 捷，并且在各种业界大会上，沾沾自喜地分享他们的经验，所以我后来也很少参加敏捷的这些会了。我怀着一颗敬畏的心，去读大师的作品，思考我工作中的不足。 我感觉自己做的工作都很普通，都是大师们说过的，很普通的那些事情，值得分享的不太多，失败倒是有很多。

技术人攻略：你从什么时候成为技术圈的交际花？你工作跨的圈子很多，是否会影响知识的积累？

我也不知道从什么时候开始成为交际花，大概是2011年初，我翻译了《MongoDB权威指南》那本书之后，参加了很多技术交流活动。在这些活动上，遇到了不少志同道合、真正热爱技术的人，让我感觉很踏实，于是就更愿意参加社区活动。

客观地说，许多活动的内容和组织都还有很大的提升空间。大家感觉日本的技术做得应该不怎么好，但我最近看了一本日本的技术刊物，发现他们的技术 讨论很深入，国内罕有。我之所以会去各个技术会议当出品人和主持人，是因为不想作为批评者旁观，而是主动推动这些事情的发展，促成技术交流的气氛。

离开上一家公司之后，许多做互联网金融的公司都给我打电话。我会问他们一个问题：技术能给你们公司带来什么？在我看来，大部分互联网金融

公司处在初期阶段，还远远没到拼技术的时候。他们拼的都还是业务，业务做得好，技术外包一个，都能撑过去。

现在加入的这家公司做APM应用性能监控，提供的是纯技术产品。我个人更希望在一家不浮躁，纯粹以技术为价值导向的公司工作。云计算快速发展，已经到了踏踏实实给大家创造价值的阶段，我希望通过公司提供的SaaS，让大家用得更舒服、更省钱。

我一直很喜欢跨领域学习，对很多东西都有好奇心。大学本来是航天工程力学系的，却因为研究工业自动化系统，获得了电气工程学院的Rockwell奖学金。尝试新东西在我看来不是障碍，而是乐趣。

工作以后，在不同圈子认识了不同的人，在互联网金融圈认识的人，如果一直待在广告圈，是无论如何也遇不上的。在各个圈子有朋友之后，可以做一些 融合的事，比如想知道金融里面安全怎么做，就会有很多安全圈的朋友给我出主意。当然，在不同圈子太多，单一的业务上说不上特别精通，但我个人积累的重心一 直放在技术上，一直在认真研究和探讨自己感兴趣的东西，从来没有放弃过对积累的追求。

作者介绍:

技术人攻略访谈是关于技术人生活和成长的系列访问,由独立媒体人Gracia创立和维护。报道内容以“人”为核心，通过技术人的故事传递技术梦想； 同时以小见大，见证技术的发展和行业的变迁。在这个前所未有的变革时代下，我们的眼光将投向有关：创造力、好奇心、冒险精神，这样一些长期被忽略的美好品 质上。相信通过这样一群心怀梦想，并且正脚踏实地在改变世界的技术人，这些美好的东西将重新获得珍视。

原文链接：http://segmentfault.com/blog/devlevelup/1190000000749805